Theoretical Linguistics Programme, Budapest University (ELTE)

# STRONG COMPOSITIONALITY

László Kálmán

Research Institute for Linguistics, Hungarian Academy of Sciences

Working Papers in the Theory of Grammar, Vol. 2, No. 3

Received: June 1995

# STRONG COMPOSITIONALITY

László Kálmán

Theoretical Linguistics Programme, Budapest University (ELTE)
c/o Research Institute for Linguistics, HAS, Room 119
Budapest I., P.O. Box 19. H-1250 Hungary
E-mail: kalman@nytud.hu

Working Papers in the Theory of Grammar, Vol. 2, No. 3
Supported by the Hungarian National Research Fund (OTKA)

**Theoretical Linguistics Programme, Budapest University (ELTE)**
**Research Institute for Linguistics, Hungarian Academy of Sciences**

**Budapest I., P.O. Box 19. H-1250 Hungary**
**Telephone: (36-1) 175 8285; Fax: (36-1) 212 2050**

## 0. Introduction

This paper addresses the much-debated issue of the *Principle of Compositionality*, the exact content, the formulation and the validity of which has been questioned so often. In its weakest form, it requires the interpretation of natural-language utterances to assign meanings in a more or less systematic manner. In Montague's (1970) classical version, it stipulates a perfect parallelism between semantic and syntactic structures. Its validity has been questioned on the basis of the influence of the (utterance-external and utterance-internal) context on interpretation. Finally, it also has acquired an *intuitive* meaning, which roughly corresponds to minimizing idiomaticity.

In this paper, I will argue that the principle of compositionality has several weak points, which make its actual content far weaker than its intuitive interpretation. As a matter of fact, the principle is almost vacuous in its popular form. **Section 1** introduces the principle and explains what its weak points are. I conclude that Compositionality must be strengthened if it is to act as a more substantial constraint on interpretation. I will also explain how the intuitive idea that idioms are exceptions to Compositionality can be made more technical.

**Section 2** presents a language that is prototypically non-compositional in the intuitive sense. The language in question is the command language of the Unix operating system. After explaining the basics of the language (**Section 2.1**), I will argue that it is no accident that users have difficulty mastering it. Although its interpretation is perfectly compatible with the popular form of the principle of compositionality, it is fundamentally different from natural languages in ways obviously related to the intuitive concept of compositionality as described in the first section (**Section 2.2**).

**Section 3** is the core of the paper. I propose two additional constraints on the interpretation of natural-language expressions: *Independence* says that the meanings assigned to sub-expressions in an expression must not depend on each other's *shapes* (**Section 3.1**); and *Additivity* prohibits operations combining meanings from *destructively* modify any previously assigned meanings (**Section 3.2**). I also sketch an interpretation mechanism for Unix commands that respects *Strong Compositionality*, i.e., the conjunction of Compositionality, Independence and Additivity. **Section 3.2.1** deals with the particular problem of treating *default* mechanisms without violating Additivity.

**Section 4** intends to draw some consequences of Strong Compositionality for the semantics of natural languages. In particular, in **Section 4.1** I argue that the mathematical metaphor of 'incomplete' expressions seen as *functors* must be abandoned in terms of the principle of additivity. Instead, I will propose to abandon the concept of 'semantic incompleteness' altogether, so that the combination of meanings yields just more complete meanings from less complete ones. **Section 4.2** is about the expression-internal interactions of meanings. I argue that

such interactions are not excluded by Independence (which only bans interactions in terms of formal properties), nor by Additivity (inasmuch as the interactions are not destructive). I sketch possible analyses of phenomena in which such interactions are involved. In particular, I argue that the meanings of adjectives and nouns in 'intersective modification' constructs are 'brought into harmony' in a non-destructive way before the 'intersection' operation is performed, and this type of analyses can perhaps be extended to other types of complex expressions, such as preposition/noun and verb/direct object constructs. In conclusion, I suggest that, by relying on the same mechanisms as we use for treating default values, the analyses are compatible with Strong Compositionality. Finally, in **Section 4.3**, I introduce an operation called *coalesced intersection*, which can be used instead of function application for combining meanings. This section is about the possible techniques that allow us to treat the meanings of syntactically complete and incomplete expressions on a pair, without type theoretic distinctions.

## 1. Compositionality

It is hard to imagine a language for which no *compositional* interpretation exists under the usual definition of compositionality:[1]

(1) *Compositionality*
The meaning of a complex expression is a function of the meanings of its sub-expressions and of the way in which they are put together.

It seems obvious that, for any complex expression whatsoever, as long as we can determine 'the way in which its constituents are put together', this criterion can be satisfied.[2] The definition in (1) has at least three loose points that are jointly responsible for this:[3]

(2) *Weak points of compositionality*
(i) the *function* used for combining the meanings of the sub-expressions can be any function at all;
(ii) there is no *a priori* limitation on what objects *meanings* can be;
(iii) in the definition above, there are no constraints as to how the meanings should be *assigned* to the various sub-expressions.

Let me now elaborate on each point in (2i–iii) above.

*Ad (2i):* There are few things a function cannot do. In particular, we can define functions pointwise so that their behaviour cannot be considered *uniform* in

[1] For the variants of this definition and their history, see Partee (1984) and Szabó (1995).
[2] Cf. Partee (1984).
[3] Partee (1984) raises the same problems.

any intuitive sense. That is, although the intuitive content of compositionality implies that, under normal circumstances, the same constituent in the same syntactic role will play the same semantic role, this is not guaranteed by the definition in (1). For example, a word like *yesterday* could easily mean 'yesterday' in some sentences and 'accidentally' in others in terms of (1). True, a word like *yesterday* could be *polysemous* in the sense that there would be *two* homonymous (homophonous) words of this shape, one of which means 'yesterday' and the other 'accidentally'. In that case, we would consider them two different expressions (the surface shapes of which incidentally coincide), and the principle of compositionality would apply to both expressions separately and independently. On the other hand, were this to be the case, our hypothetical word *yesterday* would lead to ambiguities in most cases. That is, most sentences would be ambiguous if they contained it as a constituent. But the definition in (1) would even allow a situation in which *yesterday* could unambiguously refer to 'yesterday' in some cases, and 'accidentally' in all others, which is hard to imagine in any human language. (This example will be contrasted with other types of 'ambiguities' in **Section 4.2**.)

*Ad (2ii):* There are few things we could not do with artificial meanings.[4] The simplest way of showing this is the following. Obviously, the intention of the principle of compositionality as formulated in (1) is that the meanings of complex expressions depend on the *meanings* rather than the *forms* of their sub-expressions. However, if what a 'meaning' is is not constrained, then we could assign *character strings* to certain sub-expressions as their meanings (say, character strings corresponding to their orthographic form), and have the combination function behave differently depending on what those strings are, thereby getting around an essential aspect of what (1) intends to claim. I will not have too much to say about this in the following. Note, however, that a remedy for the weakness in (2i) could also help solving (2ii): For example, if the combination functions simply cannot perform operations on character strings, then the above 'trick' is not feasible.

*Ad (2iii):* There are few things we could not do if we did not constrain the assignment of meanings to expressions. Although the principle of compositionality intends to constrain the assignment of meanings to *complex* expressions, we can still do almost anything if the assignments of meanings to *simple* expressions is left unconstrained. For example, if we could determine the meaning of a simple sub-expression depending on what other expressions occur in the complex expression (and 'the way in which they are put together'), then the meanings of complex expressions could depend on the *shape* of their constituents, which would be undesirable, as pointed out in (2ii) above.

My conclusion is that new principles, constraining the compositionality principle in (1), should be developed and adopted for the syntax/semantics interface

[4] Cf. Janssen (1983) and Partee (1984).

of natural languages. They must constrain both the class of functions that can be used for combining the meanings of the sub-expressions of a complex expression, and the way in which simple expressions are assigned meanings. The following sections will be devoted to such constraints.

Before proceeding, let me briefly digress on a problem that both the traditional concept of compositionality and the stricter version to be proposed must face. This problem is that *non-transparent* expressions by definition violate the principle of compositionality, since their meanings are unpredictable from the meanings of the words that they contain and the way they are combined.[5] There are various possibilities of dealing with such expressions.

First, we could say that they constitute exceptions from the principle (that is, we could formulate the principle in such a way that it applies to *transparent* complex expressions only). I believe this is incorrect, because many non-transparent expressions are partially compositional, and preventing the principle of compositionality from applying to them would make wrong predictions. For example, in languages that make lexical distinctions between verbs with different aspectual properties, an idiom headed by a verb that belongs to a particular aspectual class will behave accordingly. Thus, in Russian, there are no idioms headed by perfective verbs that are not perfective as a whole, and so on.

Second, we could say that non-transparent expressions are somehow not complex at all. This would also be incorrect, because most non-transparent complex expressions pattern together with transparent ones, and participate in similar processes, so that their syntactic complexity is undeniable. Now, the term *complex expression* in the principle (1) is to be understood in the syntactic sense, or the entire principle becomes vacuous.

Third, we can say that the idiomatic, non-transparent aspect of a complex expression belongs to 'the way in which the sub-expressions are put together'. I think this is the only correct way of dealing with non-transparent expressions. For example, *take the upper hand* is clearly a complex expression with a non-transparent meaning. Now assume that 'the way in which its parts are put together' is not simply the derivation that produces analogous transparent expressions, but an exceptional, qualified version of that derivation, which produces just this phrase. Then its meaning can be seen as perfectly compositional. True, this procedure would allow us to explain away many odd cases; it would even allow us to treat intuitively non-compositional languages as compositional (but having many exceptional derivations, maybe for every expression). But positing special derivations is

[5] This problem was used as a counter-argument against compositionality by Bresnan (1978), among others.

costly and calls for independent motivation. On the other hand, this treatment allows us to treat 'partially transparent' expressions correctly, since any exceptional derivation may have regular, non-exceptional aspects.

To conclude, I would like to emphasise that the problem of non-transparent expressions is unrelated to the problems of compositionality mentioned in the remarks (2i–iii) above and, in general, to the issues addressed in this paper, so the arguments given above are tangential to the main points to be made.

## 2. A Non-Compositional Language

As I suggested in the above, the sense in which natural languages are intuitively compositional is much narrower than the definition in (1). To illustrate this, and to come closer to what further constraints should be imposed, let me briefly present a language which is wildly non-compositional in the intuitive sense, i.e., very dissimilar to natural languages, although it can be given a compositional interpretation under the definition (1). The language in question is the language of *Unix commands*. As a matter of fact, probably the command language of any computer operating system would do. I chose Unix because it is best known to the academic community. The reader familiar with the basics of Unix commands may skip the following sub-section.

### 2.1. Basics of Unix Commands

A Unix command (also called *command line*) consists of a *command name*, followed by a list of *options* (which are in principle all optional and their order does not matter in principle), followed by zero or more *arguments* (some of which may be optional). (The options and the arguments together are called *parameters*.) For example, the command name `rm` (for 'remove') can be followed by the options `-d`, `-f`, `-i` and/or `-r`, and an argument ⟨file⟩ (the name of the file to be removed).[6]

Options come in two varieties: they either consist of a hyphen and a string (such an option is called a *switch* or a *flag*: the options of `rm` mentioned above belong here), or a hyphen, a string and an other string. For example, the command line

`make -f` ⟨makefile⟩ ⟨target⟩

[6] I will make unforgivable simplifications here and in the following as far as the real complexities of the Unix command language are concerned. For example, in actual fact, many other options can appear with `rm` in various implementations, and in all implementations, more than one file name can be given.

invokes the program `make` with the argument ⟨target⟩, and specifies that the rules of producing the target are described in a file called ⟨makefile⟩. The latter type of options are said to consist of an *option letter* (`-f` above) followed by its *argument* (⟨makefile⟩ above). (Whether a blank is needed between the option letter and its argument depends on the implementation, or even on the individual command names.) With some command names, the flags and the option letters can be mixed freely, and the arguments of the option letters may follow a conglomerate of flags and option letters in the order in which the corresponding option letters occur. In some cases, such a conglomerate can be written in one single word, introduced with a single hyphen. So, if the command name `make` can be followed by the flag `-n` ('do nothing, just show what you would do'), then the following may be equivalent in an implementation:

`make -n -f` ⟨makefile⟩ ⟨target⟩
`make -n -f`⟨makefile⟩ ⟨target⟩
`make -nf` ⟨makefile⟩ ⟨target⟩
`make -fn` ⟨makefile⟩ ⟨target⟩
...

As I said, options are in principle optional, but sometimes the situation is much more complex. For example, it may happen that the presence of *either* a particular argument *or* a particular option is obligatory. For example, the command name `grep` ('look for strings in a file') can be followed by an argument ⟨expr⟩ (the expression describing the strings to look for) or an option '`-f` ⟨exprfile⟩' (the name of a file in which such an expression is to be found); exactly one of the two is obligatory. In other cases, an option may appear only if another option is also present. For example, the command name `ls` ('list names, properties etc. of files') can be followed by the option `-t` ('sort by time rather than alphabetically'); an other flag, `-u`, can also appear ('sort by last access time rather than last modification time'), but only if `-t` is also present. Finally, under normal circumstances, arguments may be optional only if the remaining arguments are also optional, because arguments are only marked by their positions (they are not introduced with option letters).

For the sake of clarity, a careful distinction is to be made between the *command language* and the *shell languages* of an operating system. The command language determines how *programs* stored in the computer can be invoked with various parameters. The shells, on the other hand, are *command interpreters*, i.e., programs which help the user issue commands by preprocessing his/her input command lines. Shells usually offer interesting possibilities to the user, e.g., they remember where various programs are to be found (for a faster invocation), they remember what commands have been issued earlier (so that it is easier to

issue earlier commands again and again), they allow the user to use all sorts of abbreviations for command lines, file names etc., they help the user invoke programs one after the other, or in various combinations (for example, they contain flow-of-control possibilities like the *if... then...* construct), and they provide the user with *built-in commands* that do not correspond to separate programs, but are carried out by the shell itself. For the sake of simplicity, the language that I am talking about here is the command language rather than any particular shell language.

## 2.2. The Non-Compositionality of Unix Commands

As I mentioned in **Section 1**, the compositionality of an entire language is a gradual concept. It may happen that every single expression of a language is constructed using some special derivation. Such a language would satisfy the principle of compositionality in a vacuous way, since none of its expressions would be transparent. However, it would be entirely non-compositional in the intuitive sense, i.e., it would be very dissimilar to natural languages. The Unix command language constitutes an intermediate case: most command lines can be seen as having a more or less transparent interpretation, yet all users agree that their semantics is far from similar to the semantic of natural-language utterances.

What is the reason why Unix commands 'feel' so non-compositional at times? Obviously, this is due to the fact that, despite their fairly regular syntax (in the non-technical sense of 'regular'), expressions in the same syntactic position (even literally identical expressions) fulfil very different functions from one command to the other. That is, both the simple expressions (command names, flags, option letters) and their ways of combination (being a first or second argument, being an option etc.) *lack constant meanings*. Let me present a few examples to illustrate this.

Command names may have multiple (vaguely related) functions. For example, the command `mount` is used to attach a data-storing device (such as a data disk) to the file system when it is followed by an argument, whereas it displays information on the currently attached data-storing devices when invoked without an argument. The command `sendmail` has many unrelated functions, such as sending mail, rebuilding the database describing mail protocols, and so on.

The meanings of flags may vary from one command name to the other. For example, the flag `-l` means 'produce long, verbose listing' when used with the command `ls` (mentioned above), whereas it means 'count lines only' when it follows the command name `wc` (which is used for counting the number of lines, words and/or characters in a file). The flag `-v` means 'produce verbose output' with many commands, whereas it means 'display non-matching rather than matching lines' when used with `grep`. As we have seen, `-f` as an option letter may abbreviate

'file' (it precedes the name of an auxiliary file, such as one containing commands or expressions), but it stands for 'force' when used with `rm` (in which case various precautions are not taken by the `rm` program).

There is no uniformity as to what is expressed with an (obligatory or optional, first or second) argument or with an option. For example, as we have seen, the expression that tells `grep` what strings to look for may appear in an option (preceded by `-e`) or as an obligatory argument. The name of the target archive file of the `tar` archiving utility can only be expressed with an option ('`-f` ⟨file⟩'), while its argument is the name of the file to be extracted from or added to the archive. On the other hand, as we have seen, the name of the target is an argument of the command name `make`.

The examples could be listed indefinitely. Everyone who has used Unix will know how often the so-called *manual pages* (descriptions of the syntax and semantics of each command) have to be consulted in order to find out about the idiosyncratic properties of a command. Clearly, the heterogeneous behaviour of the commands is due to the fact that each command name corresponds to a program, and it is those programs rather than the shells or the operating system that deal with the parameters given in the commmand line. That is, it is within the discretion of the programmer who creates those programs to define how they should behave. In other words, the interpretation of the parameters is the 'internal affair' of each individual program. The only way to constrain the heterogeneity of their interpretation is to instruct the programmers to be more consistent.

However, the interpretation of the Unix command language (and of other computer command languages, for that matter) is trivially compositional under the traditional concept of compositionality, since the programs corresponding to the commands contain the definitions of the functions that they compute, and the programs themselves are interpreted compositionally. But those programs do all sorts of 'tricks', falling into each of the three classes that I mentioned in (2i–iii) in **Section 1**. That is, (i) they embody arbitrary functions (any function that a computer program can compute); (ii) they have access to their parameters in the form of character strings, which means that the compositional calculation of the meaning of a command line must rely on an odd concept of 'meaning'; and (iii) the way in which 'real' meanings are assigned to the parameters (e.g., the way in which a character string is taken to refer to a file) is not systematic (because it is also an 'internal affair' of the programs invoked).

We can see that the intuitive non-compositionality of command languages like that of Unix stems from the weak points of the principle of compositionality as described in (2) in **Section 1**. If we were to build a shell simulating a compositional interpretation for Unix commands, we should solve the general problems of compositionality.[7]

[7] We have proposed a partial solution for this in Kálmán and Rádai (1994).

## 3. Strong Compositionality

In this section, I develop an alternative to the traditional concept of compositionality. The alternative will consist in adding two sub-principles to the traditional definition (see (1) in **Section 1**), called *Independence* and *Additivity*. The resulting, more restrictive, principle will also be called *Strong Compositionality*. The following sub-sections introduce these principles.

### 3.1. Independence

I will start with the problem of assigning meanings to the sub-expressions of an expression. We have seen that the intuitive non-compositionality of Unix commands is partly due to the fact that the program invoked is free to interpret the parameters, depending on its idiosyncratic contents. The principle of compositionality (see (1) in **Section 1**) leaves it open how the sub-expressions of a complex expression are assigned meanings. To satisfy the intuitively desired requirements of compositionality in a hypothetical Unix command language, the meanings assigned to the parameters must not depend on what the command name is and what the other parameters are. That is, meanings must not be assigned in a construction-specific manner. I propose the following principle to achieve this:

(3) *Independence*
The meanings of the constituents of a complex expression are assigned independently of each other, of the way in which they are put together, and of the function that yields the meaning of the complex expression.

If a language obeys Independence, then the meaning of an expression may not vary depending on what it is a constituent of. Were we not to impose such a constraint, very similar constructs (e.g., containg the same expression in the same syntactic role) could be interpreted in heterogeneous (or even unrelated) ways. Note that this principle implies that the meaning contributions of the constituents of an expression are constant, i.e., they do not vary from one construct to the other. This means a certain *context-independence*, which many would deny. I conceive of this as a price to pay for a reasonable alternative to the traditional concept of compositionality.

The role of the external context in the interpretation of complex expressions is undeniable, but it is not a challenge to either Compositionality or Independence as long as we can reduce it to an influence on the assignment of meanings to *simple* sub-expressions, and this seems perfectly feasible (cf. Partee (1984)). On the other hand, the principle of independence also does not prevent those *meanings* that the sub-expressions are assigned from interacting with each other to produce complex meanings (cf. **Section 4.2**). We only want to exclude the dependency of the meanings of complex expressions on the formal properties (shapes) of their sub-expressions.

## 3.2. Additivity

The variability of meaning assignment is not the only reason why the meanings of command names and parameters are not constant in Unix commands. Even if we assume the principle of independence, the programs invoked by these commands may deal with the meanings of the parameters in idiosyncratic ways, because they can do anything a computer program can do. That is how, for example, the program `mount` may behave in two entirely differently ways depending on whether it is passed an argument at all. This peculiar, heterogeneous behaviour is not related to the way in which its eventual parameter is assigned a value.

To achieve a uniform behaviour of commands, we should be able to stipulate that the meaning contribution of the command name cannot be radically altered by the presence or absence of parameters, and the other way round, the meaning contribution of a parameter or a type of argument must not be radically altered by the command that it is given to. This implies a *non-destructive* way of combining constituents: whatever each constituent contributes to the meaning of the entire complex expression must be constant from one expression to the other. This can be formulated as a separate principle:

(4) *Additivity*
The function that combines the meanings of the sub-expressions of a complex expression must not destroy the information contained in those meanings.

The name 'additivity' is motivated by the fact that, if a function obeys this principle, then the meanings of the sub-expressions are simply 'summed up' in some technical sense of the word. Obviously, the definition presupposes a concept of *information content* for meanings. Usually, this kind of concept is defined by attributing an algebraic structure to the domain(s) of meanings. The structure must contain an ordering in terms of informativity, and an operation of combining pieces of information, which should not lead out of the structure. Assuming such a concept, Additivity means that the combination of two members of such a domain must yield a third member that is ordered higher than both operands in the informativity hierarchy.

For example, assume that the meaning (denotation) of the command name `rm` is a set of processes the only effect of which is the removal of some file. To combine the meaning of `rm` with the meaning of a parameter, the parameter in question must be assigned a meaning in the same domain. For example, an argument ⟨file⟩ could be assigned the set of all processes that affect the file called ⟨file⟩, and the flag `-f` could denote all processes that do not take certain precautions. Then the meaning of '`rm -f` ⟨file⟩' could be produced by taking the *intersection* of the three sets. This operation clearly satisfies Additivity if the domain of meanings is the powerset of possible processes.

In general, Additivity is always satisfied if the domain of meanings is a powerset and the only operation that may combine meanings is intersection. Note that Additivity presupposes that the meanings of the sub-expressions combined are of the same type (i.e., they are comparable in terms of the informativity ordering). This is an unorthodox requirement, which calls for further explanation and motivation. I will turn back to it in **Section 4**.

Additivity makes it very cumbersome to deal with *non-monotonicity*, i.e., phenomena in which so-called *default values* are involved. For example, if we invoke the program `cc` (which compiles source programs written in the C language into object files) without specifying what the output (executable) file is to be called, the compiler will create a file called `a.out`. We can *override* this *default* name with the `-o` option letter. Now, if we were to interpret `cc` as the set of processes that result in compiling a source file into some object file, and the option '`-o` ⟨objfile⟩' as the processes in which the output file is ⟨objfile⟩, then we get the correct interpretation of their combination by just intersecting the two denotations. In that case, however, we will not get the right result for the interpretation of `cc` invoked without a `-o` option (in which case the output file name is `a.out`). On the other hand, if we were to build the name `a.out` into the denotation of `cc`, then the intersection with that set with the denotation of '`-o` ⟨objfile⟩' would be empty (unless ⟨objfile⟩ happens to be `a.out`). The former set would contain processes in which the output file is called `a.out`, and the other would possibly contain processes in which it is called differently. So Additivity excludes those combinations of meanings in which one meaning *destroys* or *blocks* some default information associated with another. I will turn back to this problem in a moment (in **Section 3.2.1** below).

Note that the denotation of '`-o` ⟨objfile⟩' above is related to what a certain file (referred to as 'the output file' above) is called, and the denotation of `cc` must involve the same file. Obviously, the semantic object corresponding to 'the output file' in the denotation of `cc` must be a *variable* the value of which is to be ⟨objfile⟩ in terms of the denotation of '`-o` ⟨objfile⟩'. That is, we have to assume that variables are present in the semantic domains with respect to which commands are interpreted (e.g., in the states of the computer; in the Unix operating system, so-called *environment variables* can play this role). In what follows, the term 'variable' will refer to such objects rather than variables in the language of semantic representations.

### 3.2.1. Dealing With Defaults

As is clear from the above, Additivity makes it impossible to combine meanings in such a way that one meaning overrides the defaults associated with another

meaning. As an example, I quoted Unix command lines like

$$\texttt{cc -o} \langle \text{objfile} \rangle \langle \text{sourcefile} \rangle,$$

where the default object file name associated with the command `cc` can be overridden by the file name introduced with `-o`. Obviously, it is not possible to solve this problem by just designing a command interpreter (a shell) which preprocesses the user's input command lines and obeys strong compositionality.[8] In particular, a meaning that would include a piece of information like 'if there is no `-o` option in my command line, my output file name is `a.out`', which we should assign to the command name `cc`, is probably not a possible meaning.

To treat cases like the above in an additive way, we must assume more complex domains for meanings (with the appropriate ordering in terms of informativity). For example, the denotation of the command name `cc` has to contain the set corresponding to its underspecified meaning (in which the name of the output file is not specified), plus an indication on how default values can be provided if necessary. That is, I propose a *separation* of meanings from sources of default values: the denotation of a program name is to be an ordered pair $\langle S, V \rangle$, where $S$ is the set of processes corresponding to the largely underspecified meaning, whereas $V$ is some indication of how default values can be produced.[9] For the sake of simplicity, we can say that $V$ is an assignment function assigning (default) values to variables.

Technically speaking,

$$V \subseteq \text{Var} \times \bigcup_{\tau \in \text{TYPE}} D(\tau),$$

where Var is the set of variables (as semantic objects, as I explained in **Section 3.2**), TYPE is the set of types, and $D$ is the function that assigns a domain to each type. Since $V$ is a function, we have

$$\langle x, \alpha \rangle, \langle x, \beta \rangle \in V \Rightarrow \alpha = \beta;$$

so we can say

$$V(x) =_{\text{def}} \begin{cases} \alpha & \text{if } \langle x, \alpha \rangle \in V; \\ * & \text{elsewhere} \end{cases}$$

---

[8] This is a limitation that we had to face in our earlier paper mentioned in footnote 7.

[9] The two components can be seen as the *committing* vs. *deferred* information content of denotations, as explained in Kálmán (1990).

(assuming that $* \notin \bigcup_{\tau \in \mathrm{TYPE}} D(\tau)$). Obviously, we also stipulate that, if $x_\tau \in \mathrm{Var}_\tau$ (i.e., if $x$ is a variable of type $\tau$), then

$$V(x_\tau) \in D(\tau),$$

i.e., $V$ assigns an object of the appropriate type to each variable. The 'intersection' of the two pairs $P_1 = \langle S_1, V_1 \rangle$ and $P_2 = \langle S_2, V_2 \rangle$, written $P_1 \sqcap P_2$, can be defined as follows:

$$P_1 \sqcap P_2 =_{\mathrm{def}} \langle S_1 \cap S_2, V_1 + V_2 \rangle,$$

where $V_1 + V_2$ stands for the 'combination' of the valuations $V_1$ and $V_2$, defined as follows:

$$V_1 + V_2 =_{\mathrm{def}} \{\langle x, \alpha \rangle \in V_1 \cup V_2 : \bigwedge_{\beta} \langle x, \beta \rangle \in V_1 \cup V_2 \Rightarrow \alpha = \beta\}.$$

That is, we take the union of $V_1$ and $V_2$ without those ordered pairs that assign incompatible values to the same variable. The informativity ordering over ordered pairs of the form $\langle S, V \rangle$ can now be defined as

$$P_1 \leq P_2 \Leftrightarrow_{\mathrm{def}} P_1 \sqcap P_2 = P_1.$$

That is, $P_1$ is more informative than $P_2$ if its first component is a subset of the first component of $P_2$, and its valuation is a subset of the valuation in $P_2$.

Now, assuming that the first component of $P_1$ is the set of processes corresponding to the underspecified meaning of `cc`, and the first component of $P_2$ is the set of processes in which the output file is called ⟨objfile⟩, then the first component of $P_1 \sqcap P_2$ is exactly the set of processes that we want '`cc -o` ⟨objfile⟩' to denote, irrespective of what its second component contains.

As a matter of fact, default values may be *layered* in such a way that assigning a default value to a variable changes the default values available for other variables. Therefore, in actual fact, the second component, which produces default values, should be enriched. For example, it could be a set of nodes in a *default inheritance hierarchy* from which default values can be *inherited* if necessary. I will not dwell on this possibility here, because it is not directly relevant to the issue of strong compositionality.

## 3.3. Strong Compositionality

The definition of *strong compositionality* is the conjunction of Compositionality, Independence and Additivity. I submit Strong Compositionality as an alternative to the principle of compositionality.

(5) *Strong Compositionality*
The meaning of a complex expression is strongly compositional if and only if it obeys
(i) the Principle of Compositionality (cf. (1) in **Section 1**);
(ii) the Principle of Independence (cf. (3) in **Section 3.1**); and
(iii) the Principle of Additivity (cf. (4) in **Section 3.2**).

It should be clear from the above that the three sub-principles are independent. Notice how the weaknesses of Compositionality (cf. (2) in **Section 1**) are remedied by Strong Compositionality. Additivity is an answer to the arbitrary character of combination functions (cf. (2i)), and Independence constrains the assignment of meanings to simple sub-expressions (cf. (2iii)). As I mentioned earlier, the third weakness of Compositionality (i.e., the arbitrary character of meanings, cf. (2ii)) is probably harmless if the two other problems are discarded.

# 4. Strong Compositionality in Natural Language

This section deals with some consequences of Strong Compositionality on the semantics of natural language. First, in **Section 4.1**, I will examine the effect of Additivity (and Strong Compositionality in general) on common views of semantic combination and types, and I will conclude that the traditional (Fregean) metaphor of 'incomplete' linguistic expressions as *functors* and 'complete' ones as *operands* is to be abandoned. Second, in **Section 4.2**, I will say a few words how the meanings of sub-expressions can *interact* during the process of additive meaning composition to produce the meanings of complex expressions. Finally, in **Section 4.3**, I will sketch a technical solution for combining meanings without making reference to 'globally available' variable names. This will involve an operation called *coalesced intersection*, and I will also elaborate on what kind of semantic domains we might need for the interpretation of natural-language expressions.

## 4.1. Abandoning the Functor Metaphor

I have touched upon various consequences of Strong Compositionality on natural-language semantics already. In particular, I argued that the idiomatic aspects of expressions must be accounted for in terms of special ways of combination (in **Section 1**), and I argued that the influence of expression-internal contexts on interpretation are due either to genuine ambiguity or underspecification (in

**Section 3.1.1**). In this section, I will dwell on a very particular consequence of Strong Compositionality, originating from the principle of additivity. If the combination of the meanings of constituents is to be additive in the technical sense explained in **Section 3.5**, then the denotations of the immediate constituents of a complex expressions must be comparable in terms of informativity. Were they not, we could not perform intersection-like operations on them. This implies that the traditional *functor/operand metaphor* of combining meanings must be abandoned.

In the interpretation mechanism that I have proposed for Unix command lines, both the name of a command and the option attached to it denote sets of processes (or more complex structures that contain them as components). Note, however, that an option on its own is not *complete* in the syntactic sense: no option occurs without a command name. In the traditional functor/operand approach, an incomplete expression must be a functor, which can be completed by providing it with the operands that it expects.[10] That is, options should be translated as functions expecting denotations of command names and yielding denotations of command lines. But, in general, functors are of different types than their operands. So Additivity seems to exclude the functor/operand metaphor.

Abandoning the functor/operand metaphor raises two problems:

(6) *Problems with Additivity*
  (i) syntactic obligatoriness cannot be expressed in terms of semantic incompleteness; and
  (ii) semantic incompleteness cannot be expressed in terms of functional types (expecting arguments).

The problem in (6i) does not seem too big a price to pay for Additivity. In cases like that of command names and options in Unix commands, the obligatoriness of a command name in command lines has to be stated somehow, anyway, and the fact that options do not occur on their own will follow from that statement. As a matter of fact, in most natural languages, adjuncts can occur on their own as utterances.

The problem in (6ii) looks more serious at first sight. How can we tell from the denotation of, say, a verb, if it is complete without 'intersecting' it with the denotation of an argument? What is it in the denotation of a determiner that makes it so incomplete that determiners seldom occur on their own in natural languages? The functor/argument metaphor explains this type of facts very straightforwardly, to the extent that it seems almost unquestionable. For example, if a verb expresses a relation between two entities, then it is only natural that its occurrences are incomplete unless it is complemented with two other expressions, denoting entities.

---

[10] The idea that syntactically incomplete expressions are to be considered functors originates from Frege (1870).

I propose a radical solution to this problem. Maybe 'semantic completeness' is not an indispensable concept at all. If we are ready to accept a model in which meanings are ordered in terms of informativity, it is not clear at all whether we have to posit the existence of 'complete' meanings on formal, ontological or linguistic grounds. In formal terms, it is certainly possible that the algebraic structure of meanings is not atomic, i.e., there need not be any meanings in the structure that can only be enriched in such a way that a contradiction arises. Although it is not ontologically implausible that certain entities in the model are 'complete', it is not at all clear whether any linguistic expression, even a large piece of discourse, can successfully denote such an entity. In sum, I see no compelling reason why the Fregean theory, in which sentences and individual names are 'complete', should be adopted.

In terms of this radical solution, a transitive verb with missing arguments or a determiner without a noun are never *semantically*, but at most *syntactically* incomplete. This need not imply a 'mismatch' between semantic and syntactic structure, however. In the same way as a Unix command line must start with a command name syntactically, and is systematically associated with a type of meanings, uttering a transitive verb phrase may syntactically require the utterance of a transitive verb and a direct object, and denote a certain type of states of affairs. If there is any 'mismatch' at all between syntactic and semantic structures (from this perspective), it is between the 'completeness' properties of certain syntactic constructs and their semantic counterparts: 'incompleteness' may make sense for some syntactic constructs, but not for the corresponding semantic objects.

## 4.2. Interaction of Meanings

Although Independence prohibits the meanings assigned to sub-expressions from depending on each other, there are clear cases when the meanings of sub-expressions *interact* in the process of interpretation. For example, consider the following expressions:

(7) *Uses of coffee*

a. *some ground coffee*
'some ground coffee (seeds)'

b. *a hot coffee*
'a hot coffee (liquid)'

c. *a quick coffee*
'a coffee prepared/consumed/... quickly'

d. *after a coffee*
'after consuming a coffee'

These expressions illustrate what we might call *productive ambiguity*: it is very common for names of plants (like *coffee*) to stand for their consumable parts (like

coffee beans) as well as its derivatives in various stages of preparation (like the roasted seeds and the liquid in (7a–b)). On the other hand, nouns referring to food quite often take modifiers that refer to their preparation or consumption (as *quick* in (7c)), and the nouns themselves may stand for the consumption of the food in question (as *coffee* in (7d)). If, however, we assigned the meanings 'quickly prepared' or 'quickly consumed' to *quick* in (7c) or the meaning 'consumption of coffee' to *coffee* in (7d), we would violate Independence. It is also clear that we would 'miss generalizations' if we were to treat the ambiguities in (7) as accidental surface coincidences (homonymy). On the other hand, since the formal properties of the sub-expressions play no role in the interpretation of the above examples, it must be possible to explain 'productive ambiguities' of this sort without violating Independence.

Quite obviously, languages like the Unix command language do not exhibit phenomena like the 'productive ambiguities' in (7), because these phenomena stem from the important role of *implicitness* in natural-language interpretation. The examples in (7) are compact expressions corresponding to more complex meanings, which no competent speaker would have any trouble to paraphrase. On the other hand, there is some *non-determinism* in the interpretation of such compact expressions. For example, it is not absolutely excluded for *hot coffee* in (7b) to stand for 'hot coffee powder' or 'hot coffee beans' in certain contexts.

What the examples in (7) show, then, is that natural-language meanings can be combined in more than one way, and how exactly the hearer is supposed to proceed is often left *implicit* by the speaker. Both the fact that competent speakers can produce equivalent, more explicit paraphrases and the fact that the actual choice of the paraphrase is not entirely determined indicate that the processes involved are similar to other cases related to implicit information. For example, it is usually left implicit why two sentences are put one after the other in a piece of discourse (because they are part of the same story, they support the same argument, etc.). Similarly, definite descriptions are usually compact descriptions that can be made more explicit by attaching relative clauses to them. In sum, the process of interpreting expressions like those in (7) involves something very similar to certain *discourse* processes in which the speaker expects the hearer to establish 'missing links' such as anaphoric and rhetorical relations.

This suggests that combining the meanings of natural-language expressions may involve more than the simple 'intersection' operation that I have proposed in connection with Unix commands. The meanings assigned to the constituents of such expressions are processed and brought into harmony with each other before 'intersecting' them. We can think of this 'pre-processing' as analogous to those phonological processes (e.g., assimilation) which affect the lexical phonemes when they enter into contact through affixation. Most importantly, Additivity requires

the pre-processing operations to be non-destructive. For example, in the expressions in (7), it must be possible to derive the meanings of *coffee* (and *quick*) from more *abstract* (less specific) meanings. That is, using the phonological metaphor, phenomena analogous to genuine morphological (not phonologically motivated) stem or affix alternation are excluded from semantics.

What does the difference between Unix commands and natural-language expressions lie in? As I said earlier, in **Section 3.2**, an analysis of Unix command lines that respects Additivity must posit *variables* that both command names and options have access to. For example, the definition of `cc` makes reference to the variable corresponding to 'the output file' (called, say, `OUTPUTFILE`), and the meaning of the option '`-o` ⟨objfile⟩' assigns a value to the same variable. I also mentioned, in **Section 3.2.1**, that the definition of `cc` may assign a *default value* to the same variable, which can be retrieved if necessary. What I have not dealt with so far, though, is *how* and *when* such default values are assigned. In practice, it is the program `cc` which, when invoked, checks for relevant parameters and assigns default values if none is present. Be it as it may, we can think of this process as if a program different from `cc` was invoked, one in which the value of the variable `OUTPUTFILE` is set to its default value. Such a different `cc` program would qualify as an *instance* of the original, as its denotation would be *more informative* than that of the original.

It is clear that, since the interpretation of Unix commands is the 'internal affair' of the programs that they invoke, the interpretation process must lack a mechanism of the above sort, i.e., a module that would deal with the assignment of default values by creating or invoking various instances of programs. On the other hand, we are free to posit such mechanisms for the interpretation of natural-language expressions if we can justify them. For example, we could say that the default value of the direct object of the verb *leave* is the reference location:

(8) *Default direct object of leave*
  a. *I haven't seen him. Maybe he left already.*
    'Maybe he left here already'
  b. *He went to Paris, but he left already.*
    'He left Paris already'

We could also say that some interpretation mechanism converts the meaning assigned to *leave* into an instance of that meaning, in which the variable corresponding to the direct object is assigned the reference location as a value. As this qualifies as an instantiation, it does not contradict Additivity (no information gets destroyed).

The instantiation mechanism has nothing to do with Independence, which only constrains the context-dependency of meaning assignment. So the instantiation of meanings is free to be sensitive to the internal context. That is, a strongly

compositional analysis of 'productive ambiguities' (cf. (7)) would be feasible if only we could do it in terms of this kind of mechanism.

Obviously, the examples in (7) are not similar to the working of the `cc` command in Unix. For example, it would be difficult to argue that the denotation of *coffee* refers to a variable that the denotation of *quick* also mentions. In particular, we do not want to say that the denotation of *coffee* is necessarily related to a particular process with regard to which 'quick' makes sense at all (cf. the examples (7a–b), which show no trace of any process). On the other hand, by relying on the (lexical or encyclopedic) knowledge that *coffee* is prepared and drunk by people, such processes are somehow *licensed.* The licensing process can be analogous to what the instantiation would do to the program `cc` in our hypothetical Unix command interpreter. That is, we can assume that the denotation of *coffee* contains some indication on how to assign default values to a variable that expresses 'what we do with coffee', and the instantiation process assigns one of the default values. (The existence of multiple default values is a separate, unrelated issue.) Then *quick* may modify that process in the same way as '`-o` ⟨objfile⟩' does to 'the output file' with `cc`. This is why *a quick window* does not seem to make much sense unless maybe in a workshop where the relevant variable can be assigned a default value (i.e., there is a typical process affecting/creating/... windows, as in a carpenter's workshop).

The other examples in (7) can be explained in an analogous manner. We can assume that, in the prototypical ('intersective') adjectival modification structures exemplified in (7a–b), the adjective usually modifies a relevant part of what the noun denotes, as in *pink grapefruit* 'pink on the inside' vs. *pink apple* 'pink on the outside' (cf. Quine (1960), Partee (1984)). So, in these cases, it seems that we have to instantiate the denotation of the adjective rather than that of the noun.[11] What part (or, in the case of *coffee*, what stage) of an object is relevant for the adjective to modify depends on the noun, but yields a specialized denotation of the adjective (which often lexicalizes as such, like adjectives referring to colours of human skin or hair in many languages). Finally, *after coffee* in (7d) makes it necessary to instantiate the denotation of *coffee* (so that a process affecting the coffee appears in it explicitly). The analogy of (7c) and (7d) is also shown by the fact that the same class of nouns can occur in both contexts (after *quick* and *after*).

For some other examples of interactions of meanings, let me have a quick look at verbs and their arguments. The phrase *eat the grapefruit* usually is not

[11] The difference between the interpretation of various types of adjective/noun constructs is not predicted by Keenan's (1974) 'functional principle', which says that the interpretation of a functor may depend on its operand, but not vice versa. This should not bother us at all, since we have just abandoned the functor/operand metaphor.

interpreted as eating the fruit without leaving anything from it, but as 'eating the edible part of it'. As a matter of course, the definite description in this paraphrase is almost always left implicit, to the extent that we can almost think of it as part of the lexical meaning of *eat*. But in many cases the relationship between a verb and a certain type of argument varies depending on the argument. For example, consider:

(9) *Ambiguity of bake*

a. *I baked a cake.*
'I created a cake by baking'
#'I subjected a cake to dry heat on a hot surface'

b. *I baked a potato.*
#'I created a potato by baking'
'I subjected a potato to dry heat on a hot surface'

The relation between *bake* and its direct object is one of *creation* in (9a), whereas it is a relation of *affection* in (9b). We could take either *bake* or the grammatical relation 'direct object of' to be ambiguous, and let the implausible interpretations (marked with a '#' above) be filtered out by some mechanism. On the other hand, we could also think of the two cases in (9a) vs. (9b) as triggering two different instantiations of the denotation of *bake*. In the 'correct' interpretation of (9a), the abstract meaning of *bake* is enriched in such a way that a variable corresponding to the object produced is explicitly present in it, and the interpretation of 'direct object of' is perfectly willing to affect such objects, just like the option letter `-o` is willing to affect output files. In the same way, in the preferred interpretation of (9b), we produce an instance of 'bake' in which the variable that we explicitly introduce is the main ingredient of the food prepared, and the relation 'direct object of' is again willing to take it to be the object most directly affected. The concept of 'explicit introduction' will be clarified to some extent in the next section.

I am aware that the analyses presented in this section are much too sketchy and far from complete. Remember that their main point is simply that we need not violate either Independence or Additivity to account for the interactions of *meanings* if we can think of the processes involved as *instantiation*, i.e., specification operations, which lead to more informative denotations.

## 4.3. Coalesced Intersection

In **Section 3.2** we have seen how the meanings of the command name `cc` and the option '`-o` ⟨objfile⟩' can be combined through intersection: `cc` denotes the set of computations that compile a C source file into some object file, namely, the one referred to by the value of some distinguished variable, say, `OUTPUTFILE`, and '`-o` ⟨objfile⟩' denotes the set of processes in which the value of `OUTPUTFILE` is set to ⟨objfile⟩ for the time of the computation. Obviously, the intersection

of these two meanings yields the expected result because they both refer to the variable OUTPUTFILE. We cannot always proceed in this way, however, because it is not always the case that the meaning of each constituent specifies how exactly it contributes to the meanings of complex expressions. For example, the constituents of '-o ⟨objfile⟩' are -o and ⟨objfile⟩, and the denotation of the latter does not indicate what role the file that it refers to will play in complex meanings. It is the fact that it acts as an argument of -o that determines its role. Thus, it would be tempting to consider -o a functor and ⟨objfile⟩ its operand. But we said earlier that it is undesirable to allow for function application as a way of combining meanings, so a different solution is called for.

The solution I propose is to introduce *ad hoc* distinguished variables in such cases, thereby imitating the 'normal' way of combining meanings seen above. To introduce such 'local' distinguished variables, we will use the $\lambda$-notation, so that the meanings of -o and ⟨objfile⟩ will be represented as follows:

(10) *Meanings with local variables*
   a. *Meaning of* -o*:* $\lambda x[\texttt{OUTPUTFILE} = x]$.
   b. *Meaning of* ⟨objfile⟩*:* $\lambda y[y = \langle\text{objfile}\rangle]$.

Here the square brackets abbreviate that we are talking about a set of computations during which the expression(s) that they enclose hold. That is, the meaning of -o is a function from the values of $x$ to the processes in which the value of OUTPUTFILE is identical to the value of $x$, and the meaning of ⟨objfile⟩ is a function from the values of $y$ to the set of processes in which the value of $y$ is ⟨objfile⟩. Obviously, although the above meanings are functions, we are mainly interested in their *ranges* (co-domains), i.e., the sets of processes that their bodies denote. On the other hand, the simple intersection of the two sets of processes would not yield the desired result, i.e., the meaning of '-o ⟨objfile⟩': it would give us the set of processes in which OUTPUTFILE is assigned *some* value, and ⟨objfile⟩ is the value of *some* variable. What we want is a set of processes in the values of both functions applied to *the same* argument, which ensures that OUTPUTFILE is assigned a value identical to ⟨objfile⟩. The operation that produces this set will be called *coalesced intersection*. The coalesced intersection of the meanings of -o and ⟨objfile⟩ above must be something equivalent to

(11) *Meaning of* '-o ⟨objfile⟩'
   $\lambda z[z = \texttt{OUTPUTFILE} = \langle\text{objfile}\rangle]$.

As can be seen, the basic equivalence that we want to hold for coalesced intersection is the following (the symbol '$\oplus$' stands for coalesced intersection):

(12) *Basic equivalence for Coalesced Intersection*
   $\lambda x(\varphi) \oplus \lambda y(\psi) \equiv \lambda z(\varphi[x/z] \wedge \psi[y/z])$

whenever $z$ does not occur free in either $\varphi$ or $\psi$. (As usual, $\varphi[x/z]$ is the same as $\varphi$, with the free occurrences of $x$ replaced with $z$.) To ensure this equivalence, we need to define the semantic value of coalesced intersection as follows:

(13) *Coalesced Intersection*
$[\![f \oplus g]\!]_v =_{\text{def}} \{\langle \alpha, \beta \rangle : [\![f]\!]_v(\alpha) \cap [\![g]\!]_v(\alpha) = \beta\}$,

where $[\![\cdot]\!]_v$ is the semantic-value function that assigns denotations to expressions for any assignment $v$.

We can prove that the equivalence in (12) holds using the standard definition of the $\lambda$-operator, and taking conjunction to mean intersection:

(14) *Proof of (12)*

1. $[\![\lambda x(\varphi)]\!]_v =_{\text{def}} \{\langle \alpha, \beta \rangle : [\![\varphi]\!]_{v[x:\alpha]} = \beta\}$,
   where $v[x : \alpha]$ is the same as $v$ except that it assigns $\alpha$ to $x$:

$$v[x : \alpha](y) =_{\text{def}} \begin{cases} \alpha & \text{if } y = x; \\ v(y) & \text{otherwise.} \end{cases}$$

2. $[\![\lambda x(\varphi) \oplus \lambda y(\psi)]\!]_v = \{\langle \alpha, \beta \rangle : [\![\lambda x(\varphi)]\!]_v(\alpha) \cap [\![\lambda y(\psi)]\!]_v(\alpha) = \beta\}$
   by the definition in (13), which is the same as
   $\{\langle \alpha, \beta \rangle : [\![\varphi]\!]_{v[x:\alpha]} \cap [\![\psi]\!]_{v[y:\alpha]} = \beta\}$
   by the definition in (14.1) above. This is the semantic value of the left-hand side of the equivalence in (12).
3. $[\![\varphi[x/z]]\!]_v = [\![\varphi]\!]_{v[x:v(z)]}$
   if $z$ does not occur free in $\varphi$. This is trivial.
4. $[\![\varphi[x/z] \wedge \psi[y/z]]\!]_v = [\![\varphi]\!]_{v[x:v(z)]} \cap [\![\psi]\!]_{v[y:v(z)]}$
   by the interpretation of conjunction and using (14.3) above.
5. $[\![\lambda z(\varphi[x/z] \wedge \psi[y/z])]\!]_v = \{\langle \alpha, \beta \rangle : [\![\varphi[x/z] \wedge \psi[y/z]]\!]_{v[z:\alpha]} = \beta\}$
   by the definition in (14.1) above. This is the same as
   $\{\langle \alpha, \beta \rangle : [\![\varphi]\!]_{v[z:\alpha][x:v(z)]} \cap [\![\psi]\!]_{v[z:\alpha][y:v(z)]} = \beta\}$
   by (14.4) above. Now, using the definition of $v[x : \alpha]$ in (14.1) above, it is easy to see that

$$[\![f]\!]_{v[z:\alpha][x:v(z)]} = [\![f]\!]_{v[x:\alpha]}$$

   if $z$ does not occur free in $f$. Since we have assumed in (12) that $z$ does not occur free in either $\varphi$ or $\psi$, we can use this equivalence to reduce the semantic value of the right-hand side of (12) to
   $\{\langle \alpha, \beta \rangle : [\![\varphi]\!]_{v[x:\alpha]} \cap [\![\psi]\!]_{v[y:\alpha]} = \beta\}$,
   which is identical to the semantic value of the left-hand side, as can be seen in (14.2). Q.E.D.

As a matter of fact, we can even use coalesced intersection for combining meanings which could also be combined with simple intersection. I will assume that $\lambda$-abstraction can be part of the 'harmonization' process mentioned in **Section 4.2**. So the meaning of the command name cc, which is a set of computations, can be converted into something equivalent to

(15) *Meaning of* cc
$\lambda x[\text{cc}' \wedge \texttt{OUTPUTFILE} = x]$,

which can be combined with the meaning of '`-o` ⟨objfile⟩' (see (11) above) using coalesced intersection:

(16) *Meaning of* '`cc -o` ⟨objfile⟩'
$\lambda x[\texttt{cc}' \wedge \texttt{OUTPUTFILE} = x] \oplus \lambda z[z = \texttt{OUTPUTFILE} = \langle\text{objfile}\rangle] \equiv$
$\equiv \lambda u[\texttt{cc}' \wedge u = \texttt{OUTPUTFILE} = \langle\text{objfile}\rangle]$.

Obviously, in terms of the definition in (13), coalesced intersection is defined for any two functions that assign sets to the same type of entities. If the domain of denotations is ordered in terms of informativity, and an intersection-type operation is defined for its elements (i.e., we have an operation that produces the joint information content of two elements), as we have assumed, then we can generalize coalesced intersection to functions which map to that domain. Therefore, we must be able to use coalesced intersection as the basic operation for combining natural-language meanings as well.

The semantics of Unix commands is much more complex than I have sketched so far. In an earlier paper, we proposed ordered quadruples to characterize sets of processes in a radically simplified model;[12] if we add default mechanisms (cf. **Section 3.2.1**), the situation becomes even more complex. But, for obvious reasons, the semantics of natural language has enormous complexities when compared to Unix commands.

The *semantic domains* that underlie the interpretation of natural languages are much more complex than those in the Unix command language. While the latter consists of 'machine states' (basically, file structures, in which each file has certain attributes, such as its name and the character string it contains), the models that we need for interpreting natural languages include various *possible worlds* (hypothetical or real, actual or past/future) in which various *individuals* (and, eventually, groups of individuals, if they have properties that are not predictable from those of their members) exist, various *relations* hold for them. These possible worlds are also *dynamic* in the sense that they may also change in time, which corresponds to the various *eventualities* that we can talk about in natural languages. (This type of dynamism is not to be confused with the one to be touched upon promptly.)

In modern formal semantics (called *dynamic* semantics), natural language meanings are seen as *instructions* for the hearer to *update* his/her *information state* about entities in the model. So we can think of denotations as sets of updates

[12] In Kálmán and Rádai (1994), a set of processes is characterized by (i) an assignment function expressing the *local environment* of the computation; (ii) a set of *preconditions* for the execution; (iii) a formula describing the *maximal change* effected by the process; and (iv) an *environment change* corresponding to the list of variables the values of which may change as a result of the computation.

in the same way as Unix command lines denote sets of computations, which may change information states just like computations change machine states. But an information state is much more complex than a machine state, because it is not a complete description of a model, but some representation of what information is available about it.

Owing to all these complexities, the *deferred information* content of a natural-language denotation (if we posit such a thing, as was suggested in **Section 3.2.1**) is also much larger and much complex than what we need for the interpretation of Unix commands. It must encode a large body of linguistic and non-linguistic (scientific and cultural) knowledge that may influence interactions of meanings (cf. **Section 4.2**).

In sum, the complexities of natural-language semantics are enormous. So enormous, in fact, that they prevent me from presenting short and illustrative examples of the use of coalesced intersection, even simple examples like those that I have presented from the Unix command language. In what follows, I will try to just point at some features of the syntax/semantics interface that Strong Compositionality requires.

Just the same as in the interpretation of Unix command names and parameters, which we interpreted as functions mapping to the same domain (namely, that of sets of processes), natural-language meanings will also be functions that map to the same domain. Ignoring the complexities of that domain, let us simply think of it as a structure of pieces of information about first-order models. We can think of such a piece of information as a set of *model fragments*, which represent partial information on models compatible with the current information state.[13] What we ignore in this way includes (i) deferred information altogether (i.e., the interaction of meanings will be done by *deus ex machina*); (ii) the dynamic aspect of models (so I will assume that we are talking about static states of affairs); (iii) the possibility of talking about various possible worlds (the assumption being that the utterances are about one single possible world). For the sake of completeness, here is the definition of a first-order model:

[13] Given the way we will define it, the concept of a model fragment can be seen as a formal rendering of the concept of *situations* in Kratzer (1986), Berman (1987) and Heim (1990).

(17) *First-Order Models*

$\mathcal{M} = \langle \mathcal{U}, \mathcal{I} \rangle$ is a *first-order model* of language $\mathcal{L}$ iff

1. $\mathcal{U} \neq \emptyset$ ($\mathcal{U}$ is the *universe* of the model);
2. $\mathcal{I} \subseteq \text{Con} \times \bigcup_{\tau \in \text{TYPE}} D(\tau, \mathcal{U})$, where $\text{Con} = \bigcup_{\tau \in \text{TYPE}} \text{Con}_\tau$ is the set of *non-logical constants* in $\mathcal{L}$, TYPE is the set of *types* (to be defined below), and $D(\tau, \mathcal{U})$ is the *domain* of type $\tau$ given the universe $\mathcal{U}$. $\mathcal{I}$ is called the *interpretation function* of the model;
4. If $\langle a, \alpha \rangle \in \mathcal{I}$, and $\langle a, \beta \rangle \in \mathcal{I}$, then $\alpha = \beta$. That is, $\mathcal{I}$ is a *function*.
3. If $a \in \text{Con}_\tau$, and $\langle a, \alpha \rangle \in \mathcal{I}$, then $\alpha \in D(\tau, \mathcal{U})$. That is, $\mathcal{I}$ maps non-logical constants of type $\tau$ to an element of the domain of $\tau$.

(18) *Types*

The set TYPE of *types* is the smalles set such that

(i) $e \in \text{TYPE}$;
(ii) $t \in \text{TYPE}$;
(iii) If $\alpha \in \text{TYPE}$, and $\beta \in \text{TYPE}$, then $\langle \alpha, \beta \rangle \in \text{TYPE}$.

(19) *Domains of types*

The *domain* of a type $\tau$ given a universe $\mathcal{U}$, written $D(\tau, \mathcal{U})$, is defined as follows:

(i) $D(e, \mathcal{U}) =_{\text{def}} \mathcal{U}$. That is, $e$ is the type of *individuals*.
(ii) $D(t, \mathcal{U}) =_{\text{def}} \{\emptyset, \mathcal{U}\}$. That is, $t$ is the type of *truth values*, with the empty set and the entire universe representing *false* and *truth*, respectively. (Note that *false* and *true* coincide when $\mathcal{U}$ is empty; this may never be the case when $\mathcal{U}$ is the universe of a model.)
(iii) $D(\langle \alpha, \beta \rangle, \mathcal{U}) =_{\text{def}} {}^{D(\alpha, \mathcal{U})} D(\beta, \mathcal{U})$. That is, $\langle \alpha, \beta \rangle$ is the type of *functions* mapping from the domain of $\alpha$ to the domain of $\beta$.

Now we are ready to define what a model fragment is:

(20) *Model fragments*

If $\mathcal{M} = \langle \mathcal{U}, \mathcal{I} \rangle$ is a first order model, then $\langle U, I \rangle$ is a *model fragment* over $\mathcal{M}$(written $\langle U, I \rangle \in \mathcal{F}_\mathcal{M}$) if and only if:

(i) $U \subseteq \mathcal{U}$;
(ii) $I$ is an *interpretation function* with respect to $U$, as we have seen in the definition of first-order models (see (17));
(iii) If $a \in \text{Con}_e$, then $\langle a, u \rangle \in I$ if and only if $\langle a, u \rangle \in \mathcal{I}$, and $u \in U$;
(iv) If $p \in \text{Con}_t$, then $\langle p, \alpha \rangle \in U$ if and only if $\langle p, \beta \rangle \in \mathcal{I}$ for some $\beta$ such that either (a) $\alpha = \beta = \emptyset$ or (b) $\alpha = U$ and $\beta = \mathcal{U}$;
(v) If $P \in \text{Con}_\tau$ (for $\tau \notin \{e, t\}$), then $\langle P, \alpha \rangle \in I$ if and only if $\langle P, \beta \rangle \in \mathcal{I}$ for some $\beta$ such that $\alpha = \beta \cap D(\tau, U)$. That is, if the interpretation of a constant is a function in the domain of $\tau$, then we only keep those ordered pairs from it which also figure in the domain of $\tau$ restricted to $U$.

As a matter of course, we will need an informativity ordering over sets of model fragments. This will be somewhat more complex than the subset relation. To define it properly, we will first define the relation '$\leq_{\mathcal{F}}$' over model fragments:

(21) *Informativity of model fragments*
If $\varphi_1 = \langle U_1, I_1 \rangle$ and $\varphi_2 = \langle U_2, I_2 \rangle$ are model fragments, then $\varphi_1 \leq_{\mathcal{F}} \varphi_2$ (read: '$\varphi_1$ is *at least as informative as* $\varphi_2$') if and only if $\varphi_1$ is a first-order model, and $\varphi_2 \in \mathcal{F}_{\varphi_1}$, i.e., if $\varphi_2$ is itself a model fragment with regard to $\varphi_1$.

We can now define the relation '$\leq$' over sets of model fragments:

(22) *Informativity*
If $\Phi_1$ and $\Phi_2$ are sets of model fragments, then $\Phi_1 \leq \Phi_2$ (read: '$\Phi_1$ is *at least as informative as* $\Phi_2$') if and only if

$$\bigwedge_{\varphi_1 \in \Phi_1} \bigvee_{\varphi_2 \in \Phi_2} \varphi_1 \leq_{\mathcal{F}} \varphi_2.$$

That is, $\Phi_1$ contains at most those model fragments that figure in $\Phi_2$, but possibly less (leaving less possibilities open); and possibly $\Phi_1$ contains more informative versions of some fragments in $\Phi_2$.

It should be now possible to show that an expression that is syntactically 'incomplete', e.g., one that denotes a relation, may correspond to the same kind of entities as a 'less incomplete' one (e.g., a one-place predicate) or even a 'complete one', like a sentence or an individual name.

Let me first consider an expression that denotes a relation, like the verb *loves*. Since it is a relation, any interpretation function must assign it an entity in the domain of the type $\langle e, \langle e, t \rangle \rangle$. (That is, if $\langle U, I \rangle$ is a model fragment, then $I(loves) \in D(\langle e, \langle e, t \rangle \rangle, U)$.) If, for the sake of simplicity, we assume that the model fragments in our domain are over one and the same model $\mathcal{M} = \langle \mathcal{U}, \mathcal{I} \rangle$, then the set of model fragments corresponding to *loves* will be

$$\{\langle U, I \rangle \in \mathcal{F}_{\mathcal{M}} : I(loves) \neq \emptyset\}.$$

That is, any model fragment which makes 'somebody loves somebody' true is collected into a large set. We could also take just the least informative elements of this set, because the more informative elements are predictable from them, but it is simpler to proceed in this way.

Similarly, a one-place predicate like *sleeps* or *loves Joe* (or even *Joe loves*) can be assigned a set of model fragments in a similar way:

$$\{\langle U, I \rangle \in \mathcal{F}_{\mathcal{M}} : I(sleeps) \neq \emptyset\}$$

is the set of model fragments in which 'someone sleeps'. To show that the coalesced intersection of *loves* and *Joe* (assuming that *Joe* is the direct object of *loves*, which we will get by *deus ex machina*), produces an analogous set, let us first see what model fragments correspond to individual names. Obviously, the set for *Joe* is the following:

$$\{\langle U, I\rangle \in \mathcal{F}_{\mathcal{M}}: I(Joe) \in U\},$$

the least informative element of which has $\{j\}$ as its universe (where $\mathcal{I}(Joe) = j$), and its interpretation just holds $\langle Joe, j\rangle$ and pairs of the form $\langle P, \{j\}\rangle$ (in which $P$ is a one-place predicate that holds for Joe).

To perform coalesced intersection, we will deal with functions that map expressions to sets of model fragments. For example,

$$\{\langle u, \Phi\rangle: \Phi = \{\langle U, I\rangle \in \mathcal{F}_{\mathcal{M}}: \bigvee_{v \in U} \langle u, v\rangle \in I(love)\}\},$$

corresponds to '$\lambda x$($x$ loves someone)', and

$$\{\langle u, \Phi\rangle: \Phi = \{\langle U, I\rangle \in \mathcal{F}_{\mathcal{M}}: \bigvee_{v \in U} \langle v, u\rangle \in I(love)\}\},$$

corresponds to '$\lambda x$(someone loves $x$)'. Both of these can be produced from the set of model fragments corresponding to *love* using the semantic counterpart of $\lambda$-abstraction. We will need the latter function for performing coalesced intersection with the meaning of a direct object, e.g., *Joe*. The meaning of *Joe* is also to be converted into a function of the appropriate sort, namely,

$$\{\langle u, \Phi\rangle: \Phi = \{\langle U, I\rangle \in \mathcal{F}_{\mathcal{M}}: I(Joe) = u\}\},$$

which corresponds to '$\lambda x(Joe = x)$'. The coalesced intersection of these two functions is

$$\{\langle z, \Phi\rangle: \Phi = \{\langle U, I\rangle \in \mathcal{F}_{\mathcal{M}}: \bigvee_{v \in U} \langle v, z\rangle \in I(love)\} \cap \{\langle U, I\rangle \in \mathcal{F}_{\mathcal{M}}: I(Joe) = z\}\}$$

or

$$\{\langle z, \Phi\rangle: \Phi = \{\langle U, I\rangle \in \mathcal{F}_{\mathcal{M}}: \bigvee_{v \in U} (\langle v, z\rangle \in I(love) \wedge I(Joe) = z)\}\}.$$

As can be seen, the result is a function that is only defined for Joe, and yields the set of model fragments in which someone loves Joe. To combine this meaning with that of a subject's, we have to take this set and perform $\lambda$-abstraction again. (Remember that the shift from functions to their co-domains and back is simply a technical device to treat certain entities in the domain as distinguished. Unlike in semantic theories where type theory has an explanatory status, here it is insignificant whether we represent denotations with sets or functions that map to them.)

## References

Berman, S. 1987. 'Situation-based semantics for adverbs of quantification'. In J. Blevins and A. Vainikka, eds., *University of Massachusetts Occasional Papers 12.* University of Massachusetts, Amherst.

Bresnan, J. 1978. 'A realistic transformational grammar'. In M. Halle, J. Bresnan and G. Miller, eds., *Linguistic Theory and Psychological Reality.* MIT Press, Cambridge MA.

Heim, I. 1990. 'E-type pronouns and donkey anaphora'. *Linguistics and Philosophy* **13**, 137–177.

Janssen, T.M.V. 1983. *Foundations and Applications of Montague Grammar.* Mathematisch Centrum, Amsterdam.

Kálmán, L. 1990. 'Deferred Information: The Semantics of Commitment'. In L. Kálmán and L. Pólos, eds., *Papers from the Second Symposium on Logic and Language.* Akadémiai, Budapest, 125–157.

Kálmán, L. and G. Rádai. 1994. 'Compositional interpretation of computer command languages'. Paper presented at the Fifth Symposium on Logic and Language, Noszvaj, Hungary, August 1994; to appear in *Working Papers in Theoretical Linguistics*, Theoretical Linguistics Programme, Budapest University (ELTE), and Research Institute for Linguistics, Budapest, 1995.

Keenan, E.L. 1974. 'The Functional Principle: Generalizing the notion of 'Subject-of'. *Papers from the Tenth Regional Meeting of the CLS*, pp. 298–310.

Kratzer, A. 1986. 'An investigation into the lumps of thought'. *Linguistics and Philosophy.*

Montague, R. 1970. 'Universal Grammar'. *Theoria* **36**, 373–398. Reprinted in R. Thomason, ed., *Formal Philosophy: Selected Papers of Richard Montague.* Yale University Press, New Haven, 1974.

Partee, B.H. 1984. 'Compositionality'. In F. Landman and F. Veltman, eds., *Varieties of Formal Semantics.* Foris, Dordrecht, pp. 281–311.

Quine, W.V.O. 1960. *Word and Object.* MIT Press, Cambridge MA.

Szabó, Z. 1995. *Problems of Compositionality.* Ph.D. diss., MIT, Cambridge MA.

Previous titles in this series:

**1**/1 Brody M.: *Phrase Structure and Dependence*

**1**/2 É.Kiss K.: *Generic and Existential Bare Plurals and the Classification of Predicates*

**1**/3 Kálmán L.: *Conditionals, Quantification and Bipartite Meanings*

**1**/4 Bánréti Z.: *Modularity in Sentence Parsing:* Grammaticality Judgments by Broca's Aphasics

**2**/1 Szabolcsi A.: *Strategies for Scope Taking*

**2**/2 Rádai G.–Kálmán L.: *Compositional Interpretation of Computer Command Languages*