Optional Arguments in Abstract Categorial Grammar

Chris Blom

MSc. Thesis, 30 ECTS
Master in Cognitive Artificial Intelligence,
Utrecht University
May 2012

First supervisor	:	dr. Yoad Winter
Second supervisor	:	prof. dr. Michael Moortgat

Abstract

In this thesis I investigate semantic and syntactic properties of implicit arguments in a formal grammatical framework. Many verbs are flexible in the number of syntactic arguments they take, for example *eat* and *read* can be used as intransitive verbs (1 argument) and as transitive verbs (2 arguments). Although the number of realized arguments may differ, the number of semantic arguments can be the same, which is exhibited in the entailments such verbs show when we vary the number of arguments. For example *eat* and *read* exhibit existential entailments when used with only the subject argument. To give a formal account of optionality I show how the Abstract Categorial Grammer framework can be extended with option types to accommodate optionality, and use this extension to analyze various constructions involving optional arguments.

Preface

This thesis has been written between the spring of 2011 and 2012 as part of my masters in Cognitive Artificial Intelligence. For this thesis I was looking for a interesting subject in formal linguistics, a topic that I've first encountered during my bachelor degree, and that has been of great interest ever since.

As I had a very good experience writing my bachelor thesis under Yoad's supervision, I decided to ask him I he knew a good subject for a master's thesis. After a false start investigating the relation between ambiguity and prosody, he suggested writing it about implicit arguments in the ACG framework, which proved to be a very interesting subject.

I would like to thank some people who have helped me making this thesis into what it is: Yoad Winter, for being a great supervisor and all the advice, discussions and suggestions that made this work possible. Phillipe de Groote and Joost Zwarts for all their contributions and ideas. Michael Moortgat, for the introduction to categorial grammar and reviewing this thesis. My friends, for providing the much needed distractions. My family, for their support, who were always interested in my progress, even though they had much more serious things on their minds. Finally, I want to thank Miriam, for her advice, patience and neverending support.
Two visits to Prof. De Groote's lab (INRIA, Nancy) were subsidized by a Van Gogh grant of the French-Dutch Academy. This financial support is gratefully acknowledged.

Contents

1 Introduction

Many verbs are flexible in the number of syntactic arguments they take. For example, verbs like *eat* and *read* can be used as intransitive verbs (with 1 argument), and also as transitive verbs (with 2 arguments). The number of syntactic arguments required by certain verbs may vary, but the number of required semantic arguments can remain the same. When there are fewer syntactic arguments than semantic arguments, the unfilled semantic arguments are implicitly saturated. This implicit saturation affects the meaning of a verb. For example, for verbs like *eat* and *read*, their intransitive and transitive forms are related by an existential entailment. Intransitive *eat* is equivalent to transitive *eat* with an existential indefinite object:

(1) a. Miriam eats dessert.
    b. Miriam eats.
    c. Miriam eats something.

The entailments between these sentences are :

$$1a \Rightarrow 1b \Leftrightarrow 1c$$

In (1a) the transitive form of *eat* is used, and is provided with a subject (*Miriam*) and an object (*dessert*) as its arguments. The verb *eats* can also be used intransitively, as in (1b), in which case it only requires a subject. It seems that *eats* does not require an object, when the object of (1a) is left out we get (1b), which is also a well-formed sentence. If an argument is not required, we say it is an *optional argument.* In the case of *eat*, the object is optional, and we say that *eat* has an optional object. The number of arguments that a verb with optional arguments takes may vary. In (1a) *eat* takes two arguments, and in (1b) only one : the verb *eat* requires 1 or 2 arguments, so *eat* has both intransitive and transitive forms. Verbs which do not take a fixed number of arguments are called *ambitransitive* verbs. Verbs with optional arguments, like *eat* and *read*, are by definition ambitransitive verbs.

If the optional arguments of an ambitransitive verb are not present in the surface form, they can still be present in the semantic interpretation. Such semantic arguments without a corresponding element in the surface representation are called *implicit* arguments or *understood* arguments, because there is some covert argument that implicitly saturates the unfilled semantic slots, which affects the meaning of the sentence. The fact that implicit arguments have an effect on the meaning of a sentence is demonstrated by the equivalence of (1b) and

(1c). In (1b) there is no overt object for *eat*, yet (1b) is equivalent to (1c). In (1b) *eat* is seemingly supplied with an implicit argument which has the same meaning as *something*[1]. So an implicit argument, even though not visible on the surface, can still compose with the meaning of the verb, and an explicit argument can do so as well. How can an argument of a verb be missing and still affect the meaning of the verb, and how can a verb with an optional argument compose with its argument when it is present?
A naive solution might be replacing the implicit argument with a covert quantifier equivalent to *something*. This solution is not satisfactory however, as it gives wrong predictions when there is a quantifier in subject position:

(2) a. Everyone eats.
    b. Everyone eats something

Sentence (2a) is unambiguous, while sentence (2b) is ambiguous. In (2a) the quantifier of the implicit argument always has narrow scope, while in (2b) the quantifier in object position can have both narrow and wide scope. This raises a question: how can the narrow scope behavior of quantifiers introduced by implicit arguments be explained?
The naive solution of adding *someone* is wrong for another reason: not all verbs with an optional argument carry an existential import when the argument is missing. Verbs may have different ways of handling missing objects, for example 'verbs of caring for the body' show reflexive entailments rather that existential entailments [Levin, 1993, p. 35]. An example is the verb *dress*. When *dress* has no object, it would be more appropriate to insert *herself* or *himself* than *someone*:

(3) a. Mary dresses Bob.
    b. Mary dresses someone.
    c. Mary dresses herself.
    d. Mary dresses.

[1] However, others might disagree [Mittwoch, 1982], and claim that intransitive *eat* is equivalent to transitive eat with some indefinite *edible* object. According to them, the equivalence would be Miriam eats ⇔ Miriam eats something edible. I do not argree, consider the following sentence: John ate a brick. Most people would not consider a brick to be something edible, yet the fact that John ate it implies that John ate, and hence that John ate something. There is a difference between the transitive and intransitive forms of *eat* with respect to telicity, but this does not affect the existential entailment. Similarly, if John danced then we do not think of him as dancing on the roof, but if he did dance on the roof, then he danced. This has nothing to do with argumenthood.

The entailments between these sentences are:

$$3a \Rightarrow 3b \Leftarrow 3c \Leftrightarrow 3d$$

In (3d) it is understood that Mary is not dressing just anyone, she is dressing herself. Here the implicit object causes a reflexive entailment, which is more restrictive than an existential entailment. An optional argument does not always lead to existential import, reflexivization is also a possibility. In 2.3 I will discuss some verb classes with different types of implicit arguments.

Ambitransitive verbs with an unaccusative form[2] and verbs in passive voice can also be analyzed as verbs with optional arguments. Transitive verbs with an unaccusative intransitive form and passives both have optional arguments: verbs with a transitive and unaccusative form have an optional agent[3], and passive forms have an optional *by* phrase. While similar, there are some important differences between unaccusatives and passives, which I will explain using these examples:

(4)
a. The storm sunk the ship.
b. The ship was sunk by the storm.
c. Something sunk the ship.
d. The ship was sunk by something.
e. The ship was sunk.
f. The ship sunk.

The entailments between the sentences in 4 are:

$$4a \Leftrightarrow 4b \Rightarrow 4c \Leftrightarrow 4d \Leftrightarrow 4e \Rightarrow 4f$$

The verk *sink* has an unaccusative intransitive form (4f), as well as a transitive form (4a). In (4f) no agent is implied: it is therefore appropriate to view *sink* as an ambitransitive verb [Dixon, 2000], where the syntactic argument providing the agent (the subject) is optional.

Something similar is seen with the passive form of *sink*. With the passive form it is allowed to provide an agent with a *by*-phrase (4e),

[2]an unaccusative verb is an intransitive verb whose only argument is not associated with an agent role.

[3] For verbs that have a transitive and an unaccusative intransitive form, one could say that the theta-role associated with subject of the intransitive, the patient, is the same as the theta role associated with the object of the transitive.

but this is not required (4f), so here the syntactic argument providing the agent is optional as well. However, despite these similarities, the interpretations of the unaccusative and passive without the syntactic argument providing the agent are different, as exemplified by the equivalence of (4e) with (4d) and lack of equivalence with (4f). In (4e) it is implied that there was something or someone that did or caused the sinking of the ship. There must have been an agent that caused the sinking. The *by* phrase of a passive is similar to an Unspecified Object [4]) (UO). an agent is existentially assumed. Passives and UO verbs both use existential quantification to fill missing argument slots. This is in contrast with the unaccusative use of *sink* in (4f), where there is no such assumption. Unaccusative verbs that have a related transitive verb can be treated as verbs with an optional argument. When the optional argument is missing, the relation between the syntactic arguments and semantic arguments changes. The subject is then associated with the patient, similar to passive verbs. To explain the similarities and differences between passive and unaccusative verbs I ask: how can passives and unaccusatives be treated using optional arguments?

To answer all these questions a suitable framework is needed. As no framework I know of adequately deals with optionality, first the question of how optionality can be added to a formal semantic-syntactic framework needs to be answered.

1.1 Research Questions

The questions I introduced in the introduction, which are summarized below, are the research questions of this thesis:

Q1: How can optionality be added to a formal semantic-syntactic framework in a general way?

Q2: How can an argument of a verb be missing and still affect the interpretation of the verb?

Q3: How can a verb with an optional argument compose with its argument when it is present?

Q4: How can the narrow scope behavior of quantifiers introduced by implicit arguments be explained?

Q5: How should passives and unaccusatives be formalized using optional arguments?

[4]An Unspecified Object (UO) is a object that is not realized in the surface form and is interpreted existentially

1.2 Research Methods

I will attempt to answers these questions by giving a fragment of English featuring optional arguments. The fragment should cover examples found in the literature that are relevant to the research questions, and predict the correct entailments between such sentences. Especially [Levin, 1993] is used a source of examples.

As no framework that I know of adequately deals with optional arguments, I will explain how the Abstract Categorial Grammar (ACG) framework can be extended to do this. I also give a partial implementation in Haskell, which can be used to define and type-check fragments, and to automate calculations and pretty printing.

1.3 The Position of the topic in the broader field of Artificial Intelligence

The topic concerns syntax and semantics of natural language, which are subfields of linguistics intersecting with many subfields of Artificial Intelligence, amongst which logic, computer science and philosophy. The grammatical framework used for the fragment is based on type logic and lambda calculus, and the extensions to treat optional arguments in ACG are based on ideas from functional programming. The implementation of the fragment in Haskell also demonstrates the overlap of semantics of natural language and functional programming languages. Functional programming languages, type theory, and lambda calculus are all topics in computer science with application in AI.

1.4 Current Status of the Topic

There are many works on optional arguments of verbs. A comprehensive study of verb classes and their alternations by Levin [Levin, 1993] contains information about many examples of verbs with optional arguments. Some, [Fodor and Fodor, 1980, Bresnan, 1978], treat optional arguments of verbs by listing all possible forms in the lexicon, and use meaning postulates to establish the relation between the different rules. In other works, [Carlson, 1984, Landman, 2000], arguments are treated as adjuncts, and syntax is held responsible for determing the argument structure. And in [Dowty, 1982], optional arguments treated using relation changing rules, that derive the different forms from a single lexical entry. In recent syntactic works, [Landau, 2010], it has been argued that implicit arguments, while not visible in the surface form, are present in syntax.

1.5 An outline of the structure of the thesis

In section 2 I will explain what optional arguments are and discuss some previous accounts.
In section 3 the formal framework for optionality is defined, and some examples and explanations are given.
In section 4 I will explain how verbs with and without optional arguments can be treated in this framework.
In section 5 I will describe how to add optionality to grammars without optionality.
In section 6 I explain how optional parts can interact with ordinary components, and how to obtain equivalent non-optional grammars from optional grammars.
In section 7 I will describe the Haskell implementation and how it can be used to define grammars with optionality, type-check these grammars, and pretty print them as LaTeX files. In the conclusion I give an overview of the answers to the research questions, their relevance related to the current state of affairs, and state some ideas for further research.

2 Background to Optional Arguments

In this section I review previous treatments of implicit arguments, discuss their shortcomings and advantages, and summarize which facts an account of optional arguments should cover.

2.1 Arguments vs. Adjuncts

Some works [Landman, 2000, Carlson, 1984] have argued that implicit arguments can be analyzed as adjuncts. In this section I argue that implicit arguments should be treated as optional arguments, not adjuncts. I use the following terminology for arguments and adjuncts:

Terminology: ***Arguments*** : constituents adjacent to the head: an argument completes a head : a head without its arguments is 'incomplete'.

Terminology: ***Adjuncts*** : adjuncts modify a head, changing the meaning of the head, but are not required, a head without any adjunct is well-formed.

The following examples can be used to see if the optional object of a verb behaves as an adjunct or as an argument:

(5) a. John ate.
b. John ate *a cake*.
c. *John ate *a cake two hamburgers*.
d. Mary ordered and John ate a cake.

It is clear that the object of ate is not an adjunct because it cannot be iterated, but it also does not seem to be an argument, because it may be omitted. 5d shows that Mary ordered can be coordinated with John ate. The fact that these phrases can be coordinated and because a cake is definitely an argument of ordered implies that they have the same category and hence that ate also takes a cake as its argument.

2.1.1 Sub-categorized prepositional phrases of passives

Some interesting similarities between sub-categorized prepositional phrases of passives and optional object can be observed when investigating sentences with *by* phrases of passives instead of optional objects.

(6) a. A cake was eaten.
b. A cake was eaten *by John*.

c. *A cake was eaten *by John* by Mary.

Similar to optional objects it is clear that a *by* phrase of a passive is not an adjunct because it is not iterable. Seemingly it is also not an argument because it not required.
So far these examples suggest that optional objects and *by* phrases of passives do not behave as adjuncts. but more like arguments, with the exception that they are not required. Treating implicit arguments as optional arguments is my approach to deal with this exception. From a semantic perspective, there is also a difference between arguments and adjuncts. In the introduction I have shown that a missing optional argument affects the interpretation of a verb. For example:

(7) John read $\Leftrightarrow$ John read something.

This is in contrast with adjuncts, were such entailments are less clear.

(8) John baked the cake $\not\Rightarrow$ John baked the cake for someone.

Taking these considerations into account, we define an optional argument as a complement that is not required:

Terminology: Optional arguments

An *optional argument* is a constituent adjacent to the verb that cannot be iterated, and is not required.

(in other words: it is like an ordinary argument of a verb, but not required)

2.2 Previous Accounts

2.2.1 Chomsky: Object Deletion

In [Chomsky, 1965], Chomsky treats implicit objects using a transformational rule called *object deletion*. In this treatment verbs have a lexical parameter which specifies that certain objects may be deleted. For example, for the verb *eat* it is specified in the lexicon that if is it has *something* as an object, then *something* may be deleted. A problem with this approach is that it gives wrong predictions when there is a quantifier in subject position. According to Chomsky's account, a sentence like Everyone eats must have two readings: one where the quantifier introduced by everyone has wide scope and one where the quantifier introduced by the deleted something has narrow scope. As observed in [Fodor and Fodor, 1980], a quantifier introduced by an implicit argument always has narrow scope. The object wide scope reading is unavailable for (9b):

(9) a. Everyone eats something
$\forall x.\exists x.\textsf{\textbf{eats}}(x)(y)$ For every person, there is something he or she eats.
$\exists x.\forall x.\textsf{\textbf{eats}}(x)(y)$ There is a thing such that everyone eats it.

b. Everyone eats
$\forall x.\exists x.\textsf{\textbf{eats}}(x)(y)$ For every person, there is something he or she eats.
*$\exists x.\forall x.\textsf{\textbf{eats}}(x)(y)$

A purely syntactic account such as Chomsky's Object Deletion rule cannot predict this narrow scope behavior, because it poses no restrictions on how the understood existential can take scope.

2.2.2 Bresnan: with a Lexical Rule

In [Bresnan, 1978], Bresnan analyses verbs that can be used intransitively as well as transitively using lexical rules. She proposes that there are separate lexical entries relating transitive and intransitive case to different meanings, where the meaning of the intransitive is the same as the meaning of the transitive with the object argument bound by an existential quantifier. The interpretation of both the intransitive and the transitive form is a binary relation between entities. In her treatment, the lexical entries for *eat* would be something like this:

entry:	form:	meaning:
$\textsf{eat}_{iv}$	X `eats`	$(\exists y).\textbf{eat}_{eet}(X, y)$
$\textsf{eat}_{tv}$	X `eats` Y	$\textbf{eat}_{eet}(X, Y)$

With these entries, these meanings are assigned to the following sentences:

(10) a. John $\textsf{eats}_{tv}$ something.
$\exists x.\textbf{eat}(\textbf{john}, x)$

b. John $\textsf{eats}_{iv}$.
$\exists x.\textbf{eat}(\textbf{john}, x)$

c. John $\textsf{eats}_{tv}$ cake.
$\textbf{eat}(\textbf{john}, cake)$

d. Everyone $\textsf{eats}_{iv}$.
$\exists y.\forall x.\textbf{eat}(x, y)$

One problem with this analysis is that it does not generalize well to verbs with multiple optional arguments because a verb with n optional arguments would require 2^n lexical entries. While it does not seem that the number of optional arguments is ever high enough for this to be a problem, a large number of lexical entries that are largely the same seems very redundant. A more foundational problem, pointed out in [Fodor and Fodor, 1980], is that while Bresnan's treatment correctly predicts the equivalence between (10b) and (10a), it fails to predict the narrow scope of the existential quantifier when there is a quantifier in subject position, because there are no restrictions in how the covert quantifier might take scope.

2.2.3 Fodor & Fodor: using Meaning Postulates

In [Fodor and Fodor, 1980] Fodor & Fodor discuss Bresnan's treatment of implicit arguments of verbs that can be used transitively or intransitively and propose a new account. Rather than employing a lexical rule, they rely on meaning postulates to explain the entailments between the transitive and intransitive form. Like Bresnan, F&F assume there is a separate lexical entry for each specific instance of a verb, but instead of assuming two entries containing the same predicate they use two entries with different one entry with a 2-place predicate for the transitive form. For example, in F&F's treatment, *eat* would have two lexical entries, each using a different predicate with a different arity:

entry:	form:	meaning:
$\mathtt{eat}_{iv}$	X `eats`	$\mathbf{eat}_{iv}(X)$
$\mathtt{eat}_{tv}$	X `eats` Y	$\mathbf{eat}_{tv}(X, Y)$

Using these entries the following LF's would be assigned to these sentences:

1. John eats.
   $\mathbf{eat}_{iv}(\mathbf{john}_e)$
2. John eats pizza.
   $\mathbf{eat}_{tv}(\mathbf{john}_e, \mathbf{pizza}_e)$

Since F&F use different predicates, some mechanism is needed to explain the existential entailment between these sentences. F&F use meaning postulates to relate the predicates of the transitive and intransitive forms. For *eat*, the meaning postulate is defined as:

$$\text{for all } x \ : \ \mathbf{eat}_{iv}(x) \Leftrightarrow \exists y.\mathbf{eat}_{tv}(x, y)$$

Using this approach the existential entailments and narrow scope behavior of the existential quantifier introduced by the implicit argument are explained. While adequate, the use of multiple entries and meaning postulates makes this treatment of implicit arguments inelegant, as it also does not generalize well to constructions with multiple optional arguments, such as ditransitive verbs in passive voice. The passive of *introduce* for example would require four entries to cover all the combinations of missing and present arguments:

(11) a. John was introduced to Mary by Bob.
$\textbf{introduce}_{dtv}(\textbf{john}, \textbf{bob}, \textbf{mary})$
b. John was introduced by Bob.
$\textbf{introduce}_{tv1}(\textbf{john}, \textbf{bob})$
c. John was introduced to Mary.
$\textbf{introduce}_{tv2}(\textbf{john}, \textbf{mary})$
d. John was introduced to Mary by Bob.
$\textbf{introduce}_{iv}(\textbf{john})$

To relate the meanings of the entries four meaning postulates would be required. Using this approach it, n^2 entries and MP's are needed for each n optional arguments.

F&F mention another problem: not all ambitransitive verbs feature existential entailments. With certain verbs, such as *notice*, the implicit arguments are not interpreted as existentials, but rather as an anaphora and other verb classes feature reflexive entailments. Unaccusatives show even different entailments, here the implicit argument seems to be simply not present at all:

$$\text{for all } x, y \ : \ \textbf{sink}_{iv}(x) \Leftarrow \textbf{sink}_{tv}(x, y)$$

To account for the relevant phenomena using meaning postulates each verb class would require its own set of meaning postulates, for different numbers of optional arguments. Since many combinations and variations are possible, many closely related entries in the lexicon and many meaning postulates are required. While empirically adequate, the use of multiple entries for verbs with closely related forms and meanings seems inelegant.

2.2.4 Carlson: optional arguments as adjuncts

Carlson uses event semantics to give an account of Unspecified Objects and *by* phrases of passives [Carlson, 1984], treating both as role adding event modifiers. According to Carlson in a sentence like John ate pizza,

the object pizza would denote a modifier that adds a patient to an event.
In a sentence like Lolita was read by John, the *by* phrase by John would denote a modifier that adds a patient to an event. Carlson assumes that it is an implied property of reading events that there must be an agent who did the reading and a patient that whas read. This implied propery of events accounts for the existential import over missing argument slots.

This assumption avoids the complications of F&F when there are multiple optional arguments, as the interpretation of missing arguments is determined by a single meaning postulate. The entailments for a verb with two optional arguments, like passive *introduce*, would be covered by a single meaning postulate instead of four as in F&F:

$$\forall e.\mathbf{introduce}(e) \rightarrow \exists x \exists y.\mathsf{AG}(e,x) \wedge \mathsf{GOAL}(e,x)$$

While Carlson's approach improves upon F&F's approach when dealing with multiple optional arguments, it has some problems. Treating *by* phrases as role adding modifiers is too general when considering unaccusatives.

(12) a. The ship sank
 b. *The ship sank by John

Adding an agent role to an unaccusative verb with a *by* phrase should not be allowed (12b), but there is nothing that prevents the use of *by* phrases in combination with unaccusatives in Carlson's theory. So while Carlson's solution can deal with multiple optional arguments, it is not restrictive enough. The reason for this is that UO's and *by* phrases of passives are not adjuncts, a point which I discussed in section 2.1. Any attempt to treat optional arguments as adjuncts must require some ad hoc restrictions to avoid overgeneralisations.

2.2.5 Dowty: Relation-Reducing rules

In [Dowty, 1982], Dowty uses relation-reducing rules to explain entailments between related transitive and intransitive verbs, and active verbs and their passive counterparts.
To account for UO's and passives without a *by* phrase, Dowty uses rules that transform a transitive verb into an intransitive verb by existentially quantifying over the object argument slot of the verb predicate and adding *by* for the passive. The specific instance of rules for transitive *eat* can be paraphrased as follows:

$$(\text{ X } \textit{eat} \text{ Y },\ \mathbf{eat}_{eet}) \xrightarrow{\mathsf{object-deletion}} (\text{X } \textit{eat},\ \lambda x.\exists y.\mathbf{eat}_{eet})$$

$$(\text{ X } \textit{eat} \text{ Y },\ \mathbf{eat}_{eet}) \xrightarrow{\mathsf{agentless-passive}} (\text{X } \textit{be eat},\ \lambda x.\exists y.\mathbf{eat}_{eet})$$

Such rules explain the narrow scope existential entailments in an elegant way, but this approach suffer from the same disadvantage as the other approaches. When the number of optional arguments increases, so do the number of rules required to cover all the cases. Each specific form of an ambitransitive verb will require a different rule. Another disadvantage is that while the optional arguments of passives and transitive verbs with UO's are handled the same way (using existential saturation), two different rules are required. The idea of using rules that change both the form and meaning of word is very elegant. The rules discussed in section 5 and Dowty's rules are similar in many ways. In fact the object deletion rule Dowty gives is a specific instance of the general optionalization procedure in section 5.

2.2.6 Landman: *By* phrases as adverbials

In [Landman, 2000], Landman uses a neo-Davidsonian framework in which he treats *by* phrases of passive verbs as event modifying adverbial adjuncts. In this book, he assumes that passive verbs are derived from active verbs by existentially quantifying over the agent argument, similar to Dowty's relation reducing rules. Where Dowty uses multiple rules that map an active verb to agentless or agentive passive, Landman uses only one rule for passivisation. This rule maps a transitive verb to an agentless passive verb, by existentially quantifying over the agent argument. An agentive passive is then formed by an adjunction of a *by* phrase to an agentless passive, much like in Carlson's account. Landman treats the *by* phrase as an adjunct, which behaves as denotes a function from an entity x to a modifier of verb phrases, which sets x as the agent of the VP. This approach gives correct results with respect to the existential entailments between a transitive verb and its passive counter part, and the narrow scope of the implicit agent of an agentles passive. However, as with Carlson's account, treating a *by* phrase as an adjunct gives wrong predictions when considering unaccusatives.

2.3 Types of Optional Arguments

For certain verbs that allow optional arguments, like *eat* and *read*, the absence of the optional argument triggers an existential interpretation. With a broader conception of optional arguments, we can also consider other phenomena as optional arguments. Sub-categorized prepositional phrases of passive verbs can also be analyzed as optional arguments, whose absence also triggers an existential interpretation. Different types of verbs can have different kinds of understood arguments, which are characterized by how the absence of an overt argument affects the interpretation of the verb. Verbs like *wash* or *shave*, for example, have understood reflexive objects when the optional object is missing: if X shaves then X shaves himself. This section explains how certain verb classes, taken from [Levin, 1993], can be analyzed as verbs with optional arguments.

2.3.1 Understood Existential Arguments: Unspecified Objects and Passives

This class of optional arguments is characterized by its existential behavior: when the optional argument is omitted, it is interpreted similar to an existential quantifier, like *someone* or *something*. A missing optional argument in this class behaves like an *understood existential argument*.

(13) a. John ate $\Leftrightarrow$ John ate something.
b. John ate $\not\Leftrightarrow$ John ate nothing.
c. Everyone ate $\Leftrightarrow$ For everyone there was some X such that he or she ate X.
d. Everyone ate $\not\Leftrightarrow$ There was some X, such that everyone ate X.

As discussed in section 2.2.3 the quantifier introduced by an understood object has always takes narrow scope. I believe that subcategorized prepositional phrases of passive forms (*by* phrases) can be seen as optional arguments of the passive verb. The main argument for this is that when a *by* phrase of a passive verb is missing, their unfilled semantic slots are filled by an understood existential (14), much like how the slots unfilled by a missing object of a transitive verb are filled.

(14) a. The cake was eaten$\Leftrightarrow$ The cake was eaten by someone
b. Every dish was eaten$\Leftrightarrow$ For each dish, there was someone who ate it.

An account of optional arguments in unspecified objects and passives should correctly predict these narrow scope existential entailments. Existential optional arguments are special for two reasons: *by* phrases of passives behave like understood existential arguments, and certain verbs have multiple understood existential arguments. Other types of optionality are restricted to a single argument.

2.3.2 Understood Reflexive Object

This class of optional arguments is characterized by reflexive behavior. In this class we see verbs of caring for the whole body.

(15) a. John shaves $\Leftrightarrow$ John shaves *himself*.
b. Mary dresses $\Leftrightarrow$ Mary dresses *herself*.

When the optional argument is omitted it is interpreted similar to a reflexive, like *itself* or *himself*. In many languages these verbs do not have true intransitive form, but allow reflexive clitics in object position. When such a clitic is supplied, the meaning of this is similar to the intransitive reflexive verb in English.

An account of reflexive optional arguments in English should predict this reflexive interpretation when the argument is missing. For languages that use reflexive clitics instead, it is debatable if these are the result of missing optional arguments, or that some other mechanism is responsible. Nevertheless, a treatment involving option types that introduces reflexive clitics can be given.

2.3.3 Understood Reciprocal Object

Another class of optional arguments is characterized by its reciprocal behavior: when the optional object is omitted it is interpreted much like a covert reciprocal[5]. For example:

(16) a. John and Mary kissed. $\sim$ John and Mary kissed *each other*.
b. Terrence and Phillip battled. $\sim$ Terrence and Phillip battled *each other*.

[5] The meaning of an implicit reciprocal object is not exactly the same as that of an overt reciprocal, hence the use of $\sim$ instead of $\Leftrightarrow$. The reason for this is that an implicit reciprocal, as in (1b), requires that the participants are aware of the mutual action. Consider the following sentences:

(1) a. John kissed Mary in his sleep and Mary kissed John in her sleep
b. John and Mary kissed.
c. John and Mary kissed each other.

(1a) entails (1c), even though the participants are not aware of the mutual action, but (1a) does not entail (1b). This difference shows that an overt reciprocal, while similar in meaning, is not equivalent to an overt reciprocal.

(17) a. John kissed. $\not\leadsto$ *John kissed *each other*.
b. Terrence battled. $\not\leadsto$ *Terrence battled *each other*.

When such verbs are used intransitively, the subject must be a collective NP.

2.3.4 Understood Object

This class of optional arguments is characterized by the introduction of some understood object when the argument is missing.

(18) a. John nods $\Leftrightarrow$ John nods his head.
b. Mary blinks $\Leftrightarrow$ Mary blinks her eye.
c. Bob claps $\Leftrightarrow$ Bob claps his hand.

It seems that verbs that allow a highly restricted set of words as arguments allow the omission of that argument, and when the argument is omitted, it is implicitly interpreted as a word in this restricted set. For example, a verb like *blink* is highly restricted in its object argument: when the subject is a person, very few phrases can be used as the object. I believe this is why such objects may be omitted: since there are few possible phrases (maybe even one) that are allowed there, it is not necessary to explicitly mention the object to get the message across. An account of this kind of optional arguments in English should predict these understood argument entailments, but since these understood arguments refer to some very specific object, interesting generalizations are not likely.

2.3.5 Understood Anaphoric Object

Another class of optional arguments is characterized by anaphora-like behavior, where a missing object is interpreted like a covert discourse referent [Fodor and Fodor, 1980]:

(19) John noticed $\Leftrightarrow$ John noticed *it*.

Because the semantic framework I will use to analyze optional arguments is static, I will not discuss this class of implicit arguments. I foresee no issue in treating understood anaphora in a dynamic semantics that also supports optional arguments.

2.3.6 Unaccusatives

Under this broader conception of optional arguments, even unaccusative verbs can be analyzed using optional arguments. Some unaccusatives have a related transitive form, that is similar in meaning to the intransitive form where the extra argument specifies an agent or 'causer' of the verb event that is not specified in the unaccusative form. The transitive use of a verb *V-transitive* is similar in meaning to *cause to V-unaccusative* [Levin, 1993]. The agent argument can be understood as an argument that is optionally added to an unaccusative verb. For example, in (20a), *John* could be considered optional agent argument of *sank*, and *Mary* an optional agent argument of *opened*.

(20) a. John sank the ship. ⇒ The ship sank.
b. Mary opened the door. ⇒ The door opened.
c. The ship was sunk. ⇒ The ship sank.
d. The bullet shattered the window. ⇒ The window shattered.

When the agent argument is missing in an unaccusative form, there is no implicit saturation of an unfilled semantic slot. This is what sets unaccusatives apart from the other discussed verbs classes that allow optional arguments. For verbs like *eat*, an understood existential object is introduced when the optional object is missing. For verbs like *wash* it is an understood reflexive object. Other understood arguments are introduced for other types of verbs that have optional arguments, but for unaccusatives no understood argument is needed. This is demonstrated by the fact that there is no simple way to give an equivalent formulation of the ship sank using the transitive form of *sink*. Something sank the ship is not equivalent, as it implies that somebody or something did the sinking. No one sank the ship is also not equivalent, as it denies the existence of some agent or cause of the sinking. A reflexive interpretation is also not appropriate, as The ship sank itself entails Something sank the ship. This shows that while unaccusatives and passives may seem similar, there is an important difference. In both these constructionss the subject argument does not correspond with an agent, but with passives an agent is implicitly introduced if none is provided by a *by* phrase, while with unaccusatives there is no understood agent.

It is clear that the optionality that relates unaccusatives and their causative transitive counterparts is of a different nature than the optionality of the other verbs classes that allow optional arguments. The other verbs classes are *transitive* verbs, with an optional argument that is replaced with a covert understood argument when missing. Whether a syntactic argument fills the optional argument slot or not, the semantics must receive the same number of arguments. For these verb classes the optionality is *subtractive*: an overt argument can optionally be removed, and is then replaced with some covert understood argument. For unaccusative-causative verbs, the optionality is *additive*: an argument providing an agent can optionally be added, but when it is not provided, there are no unfilled slot in the semantic component of the verb.

The optionality of verbs with unaccusative and causatives forms is similar to the optionality of passives, as can been seen in example (20). In English, the optional addition of an argument providing the agent, which transforms the unaccusative into a causative transitive verb, also triggers a change in word order. The subject of the unaccusative form takes the place of the object in the transitive form, and the added argument becomes the subject. This change in word order resembles the change in word order of passives verbs compared to their active transive counterparts. An important diffence between the optionality of unaccusatives and passives is that an unfilled semantic slot of an unaccusatives is not existentially saturated, while an unfilled slot of a passive verb is.

2.4 Issues with earlier accounts

I have discussed some issues with existing approaches to implicit arguments. To summarize, any treatment of implicit arguments should attempt to avoid these issues:

- A treatment of implicit arguments should predict the narrow scope of implicit quantifiers.
  Quantifiers introduced by implicit arguments always have narrow scope.
- A treatment of implicit arguments should scale well to multiple optional arguments.
  Accounts based on multiple meaning postulates, multiple entries or multiple rules do not scale well when considering verbs with multiple optional arguments.

Optionality is an important feature of language, and given the issues stated above, I believe it is more appropriate to add a way of dealing with optional arguments to the compositional system than to add an extremely large number of entries to the lexicon.

Under a broader conception of optional arguments, other phenomena than unspecified objects can be treated as optional arguments. From a conceptual point of view, the following is a desirable feature:

- A treatment of implicit arguments should scale well to different types of entailments for different verb classes.
  Beside interpretation as existential quantifiers, implicit arguments may also be interpreted as: anaphora (notice), reflexives (wash), understood objects (blink)

3 ACG with Option Types

To give a formal account of the syntactic and semantic properties of implicit arguments I use Abstract Categorial Grammar (ACG). Abstract Categorial Grammar [de Groote, 2001] is a categorial grammar formalism. Categorial grammars are highly lexicalized (the behavior of words is defined in the lexicon as much as possible), emphasize compositionality (meanings of constituents are systemeatically combined to form meanings of more complex constituents). ACG makes a distinction between tecto-grammar from pheno-grammar, an idea that originates from [Curry, 1961].

In standard ACG, verbs, like all other functions, can only take a fixed number of arguments, To treat optional arguments I show how ACG can be extended with *option types*. Because of its type-theoretical foundations it is easy to extend ACG with concepts from functional programming and type theory, which is where option types were used first. For compatibility with [Winter and Zwarts, 2011], and for presentational purposes, I will use a formulation of ACG which shows the different components of a grammar in parallel as tuples, called *signs*. This presentation is similar in appearance to Muskens' Lambda Grammars [Muskens, 2001], but since these frameworks are all very similar, modulo some technical details, it does not really matter which presentation is used, as the underlying ideas are the same.

3.1 Components

In ACG, a grammar consists of different components[6]. There is a single *abstract component*, which is mapped to multiple *concrete components*. In my treatment I will use three components: abs for the abstract component, pheno for the morpho-syntactic component and sem for the semantic component. abs models the tecto-grammar: the general argument structure. pheno models the pheno-gramar: phonological manifestations of word order, case, agreements, etc. sem is used to model the meaning.

3.2 Types

Let B_x be a finite set of basic types for component x. The component is omitted when it is clear which component is used. Throughout this thesis I will use the following basic types for the abstract, morpho-

[6]Components are called *signatures* in [de Groote, 2001]

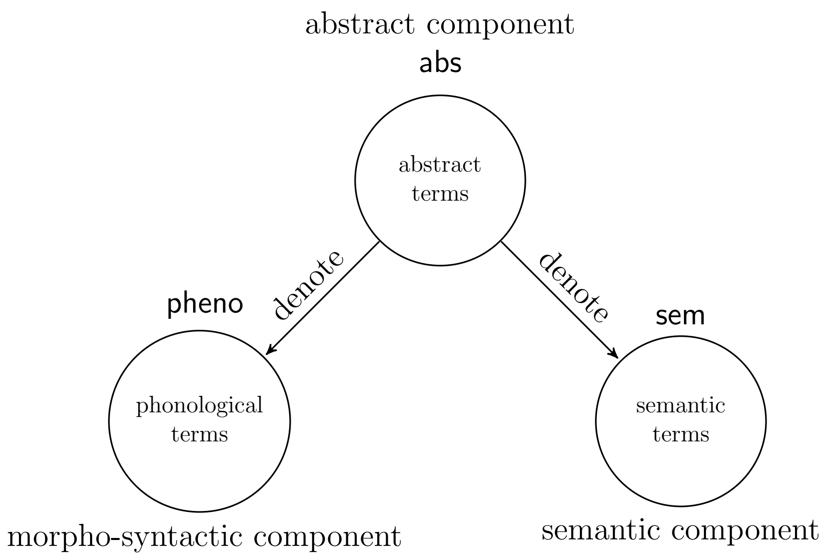

Figure 1: The architecture of the framework: a semantic and a morpho-syntactic component that share a single abstract component

syntactic and semantic components:

$$
\begin{aligned}
B_{\mathsf{abs}} &= \{\mathsf{np}, \mathsf{n}, \mathsf{vp}, \mathsf{s}\}\\
B_{\mathsf{pheno}} &= \{f\}\\
B_{\mathsf{sem}} &= \{e, t\}
\end{aligned}
$$

For the abstract component, which describes the tecto-grammar, we have the basic type np for noun phrases, n for nouns, vp for verb phrases and s for sentences. There is only one basic type available in the pheno-component: f, which is the type of strings. For the semantic component we have the standard types e for entities and t for truth-values.

The set of types is defined by combining basic types. The set of types $\mathcal{T}^{B_c}$ over the set of basic types B_c of component c is defined as the smallest set such that:

$$
\begin{array}{ll}
\text{if } x \in B_c \text{ then } x \in \mathcal{T}^{B_c} & \textit{basic types}\\
\text{if } x \in \mathcal{T}^{B_c} \text{ and } y \in \mathcal{T}^{B_c} \text{ then } (x \to y) \in \mathcal{T}^{B_c} & \textit{function types}\\
\text{if } x \in \mathcal{T}^{B_c} \text{ then } x^{?} \in \mathcal{T}^{B_c} & \textit{option types}
\end{array}
$$

I call the types over the abstract component *abstract types*, the types over the morpho-syntactic component *morpho-syntactic types* and the types over the semantic component *semantic types.*

Abstract types are typeset like this, and morpho-syntactic and semantic types like *this.* To keep things on a single page, $\rightarrow$ is omitted in morpho-syntactic and semantic types.

3.3 Type Mappings

The types of terms of concrete components all depend on the type of the abstract term. This dependency is described by the *basic type interpretation function*, which maps the basic types of one component x to types of another component y.

$$I_{x \Rightarrow y} : B_x \rightarrow \mathcal{T}^{B_y}$$

Mapping types from one component to the other is done by taking the unique homomorphic extension ($\hat{I}_{x \Rightarrow y}$ of a basic interpretation function $I_{x \Rightarrow y}$. The unique homomorphic extension of I is defined as:

$$\hat{I}(x) = I(x) \text{ if } x \text{ is a basic type}$$
$$\hat{I}(x \rightarrow y) = \hat{I}(x) \rightarrow \hat{I}(y)$$
$$\hat{I}(x^?) = \hat{I}(x)^?$$

The interpretation function maps types of component x to types of component y:

$$\hat{I}_{x \Rightarrow y} : \mathcal{T}^{B_x} \rightarrow \mathcal{T}^{B_y}$$

Throughout this thesis I will use the basic mappings from basic abstract types to concrete morpho-syntactic and semantic types defined in Figure 3 and Figure 2.

$$I_{abs \Rightarrow syn} = \begin{bmatrix} \mathsf{np} \rightarrow f \\ \mathsf{n} \rightarrow f \\ \mathsf{s} \rightarrow f \\ \mathsf{vp} \rightarrow f \end{bmatrix}$$

Figure 2: Mapping from basic abstract types to morpho-syntactic types: noun phrases, nouns, sentences and verb phrases all have strings as phonological denotations.

$$I_{abs \Rightarrow sem} = \begin{bmatrix} \mathsf{np} \rightarrow e \\ \mathsf{n} \rightarrow et \\ \mathsf{s} \rightarrow t \\ \mathsf{vp} \rightarrow et \end{bmatrix}$$

Figure 3: Mapping from basic abstract types to semantic types: noun phrases denote entities, nouns denote sets of entities, verb phrases denote sets of events and sentences denote truth values.

3.4 Terms

Let $\mathcal{T}_c$ be the set of types over basic types B of component c.
Let C_c be the set of constants typed with a type in $\mathcal{T}^c$.
Let V_c be an infinite countable set of typed λ-variables with types in $\mathcal{T}^c$
The set of typed λ-terms over typed constants C_c with types in $\mathcal{T}_c$, Λ_c is defined inductively as:

if $c : \alpha \in C_c$ then $c : \alpha \in \Lambda_c$	constants
if $v : \alpha \in V$ then $v : \alpha \in \Lambda_c$	variables
if $x : \beta \in \Lambda_c$ and $v : \alpha \in V$ then $\lambda v.x : \alpha \to \beta \in \Lambda_c$	lambda abstraction
if $m : \alpha \to \beta \in \Lambda_c$ and $n : \alpha \in \Lambda_c$ then $m(n) : \beta \in \Lambda_c$	function application
$* : \alpha^? \in \Lambda_c$	universal filler
if $x : \alpha \in \Lambda_c$ then $\overline{x} : \alpha^? \in \Lambda_c$	option injection
if $x : \alpha^?,\ y : \alpha \to \beta$ and $z : \beta \in \Lambda_c$ then $\mathsf{option}(x, y, z) : \beta \in \Lambda_c$	option analysis

In addition to the standard reduction rules of the lambda calculus, there are two additional reduction rules for the option type extensions:

$$\mathsf{option}(* : \alpha^?, f : \alpha \to \beta, d : \beta) \leadsto d : \beta$$
$$\mathsf{option}(\overline{x} : \alpha^?, f : \alpha \to \beta, d : \beta) \leadsto f(x) : \beta$$

For presentational purposes, I use different fonts for terms of different components: abstract term are typeset using Sans Serif, **semantic constants** using boldface and `strings` using the typewriter font.

The semantic component has the connectives of first-order logic as constants, which all have their usual interpretations.

$\wedge$	$: ttt$	conjunction
$\vee$	$: ttt$	disjunction
$\rightarrow$	$: ttt$	rightarrow
$\neg$	$: tt$	negation
$\top$	$: t$	true
$\bot$	$: t$	false
$\forall$	$: (et)t$	universal quantification
$\exists$	$: (et)t$	existential quantification

The notation rules for the logical constants are as follows: $\wedge$, $\vee$, $\rightarrow$ are written as infix operators, $\neg$ is written as a prefix operator. The quantifiers are normally written as prefix operators, unless their argument starts with a lambda. In that case, we use the following notation convention: if the argument of a existential quantifier has the form $\lambda x.M$ then the application of the quantifier to such a term, $\exists(\lambda x.M)$, is written as $\exists x.M$. Likewise for the universal quantifier: $\forall(\lambda y.N)$ is written as $\forall y.N$. This notation convention is used to emulate the standard notation of predicate logic.

Non-logical constants are introduced when needed. Any constant with a type built out of e's and t's can be used in the semantic component:

john : e	the entity that *john* denotes
run : et	predicate for *run*
man : et	predicate for *man*

In the morpho-syntactic domain there are two basic constants:

$\epsilon : f$	null string
$\bullet : f(ff)$	string concatenation

Other constants are introduced when needed, usually these are strings:

`john` : f	the string for *john*
`runs` : f	the string for *runs*

Strings form an associative monoid under concatenation, that is, for all $x : f$, $y : f$ and $z : f$:

$$\begin{aligned} x \bullet \epsilon &= x \\ \epsilon \bullet x &= x \\ x \bullet (y \bullet z) &= (x \bullet y) \bullet z \end{aligned}$$

For example, with the constants the following term can be constructed:

$$\texttt{john} \bullet (\texttt{runs} \bullet \epsilon)$$

Because $\bullet$ is associative the parentheses may be omitted, and because ϵ may always be omitted, the term can be reduced to:

$\texttt{john} \bullet \texttt{runs}$

3.5 Signs

Signs are tuples of typed lambda-terms, where each entry of the tuple holds a term of some component. One entry is special: this the entry of the abstract component, which we call the abstract entry. The other entries are called concrete entries. Signs are written using the following convention:

$$\textsf{abs. term} : \textsf{type} = \langle \textit{concr. term}_1 : \textit{type}_1, \ldots, \textit{concr. term}_n : \textit{type}_n \rangle$$

Signs can be combined using function application, abstraction etc., just like terms, by interpreting each operation *point-wise* over each entry. To ensure that when rules are applied they are compatible with each component, the type of each concrete entry must be compatible with the abstract type and the type interpretation function. In this setup the only two concrete components of signs are pheno and sem, but additional components could easily be added.

Let $\mathcal{V}$ be a countably finite set of variables, and A a set of typed abstract constants. The set of signs $\mathcal{S}$ is defined as:

- if $c : \alpha \in \mathcal{C}_{\textsf{abs}}$, $x : \beta \in \Lambda_{\textsf{pheno}}$ and $y : \gamma \in \Lambda_{\textsf{sem}}$
  and $I_{\textsf{pheno}}(\alpha) = \beta$ and $I_{\textsf{sem}}(\alpha) = \gamma$
  then $c : \alpha = \langle x, y \rangle \in \mathcal{S}$ constants

- if $v : \alpha \in V$ then $v_{\textsf{abs}} = \langle v_{\textsf{pheno}}, v_{\textsf{sem}} \rangle \in \mathcal{S}$ variables

- if $M_{\textsf{abs}} : a = \langle M_{\textsf{pheno}}, M_{\textsf{sem}} \rangle \in \mathcal{S}$
  and $x : b \in \mathcal{V}$,
  and x is free in all components,
  then $\lambda x.M_{\textsf{abs}} : b \to a = \langle \lambda x.M_{\textsf{pheno}}, \lambda x.M_{\textsf{sem}} \rangle \in \mathcal{S}$ abstraction

- if $M_{\textsf{abs}} : a \to b = \langle M_{\textsf{pheno}}, M_{\textsf{sem}} \rangle \in \mathcal{S}$
  and $N : a = \langle N_{\textsf{pheno}}, N_{\textsf{sem}} \rangle \in \mathcal{S}$
  then $M_{\textsf{abs}}(N_{\textsf{abs}}) : b = \langle M_{\textsf{pheno}}(N_{\textsf{pheno}}), M_{\textsf{sem}}(N_{\textsf{sem}}) \rangle \in \mathcal{S}$ application

- for all α, $* : \alpha^? = \langle *, * \rangle \in \mathcal{S}$ null option

- if $M_{\textsf{abs}} : a = \langle M_{\textsf{pheno}}, M_{\textsf{sem}} \rangle \in \mathcal{S}$
  then $\overline{M_{\textsf{abs}}} : a^? = \langle \overline{M_{\textsf{pheno}}}, \overline{M_{\textsf{sem}}} \rangle \in \mathcal{S}$ option injection

3.6 Sign grammars

A sign grammar G is a tuple of a type interpretation function, a list of signs and a distinguished abstract type d.

$G = \langle I', d, S \rangle$ where

I : a tuple of type interpretation functions

d : a distinguished basic abstract type

S : a finite set of basic signs

A sign grammar generates a single abstract language, which is the set of well-typed abstract terms of type d. It also generates concrete languages which are defined as the set of concrete terms of which the abstract term is part the abstract language and is of the distinguished abstract type.

3.7 Linearity

The system presented here is actually a simplified version of ACG: the type logic and term system are based on intuitionistic logic, while ACG is based on *linear* logic[7]. In a linear type system, the use of assumptions is more restricted: the rules of weakening and contraction cannot be used freely. At the level of terms, the use of variables is also more constrained: in a linear term, every lambda abstraction must bind exactly one variable. The linearity of the types and terms plays a critical role in placing bounds on the expressivity of the grammar, and in the tractability of parsing with ACG's. However, linearity is too strict a requirement for when modelling certain linguist phenomena. The semantics of reflexives, for example, cannot be modelled in a linear system. The denotation of a reflexive has the form $\lambda x.f(x)(x)$: single variable is used twice: once in agent position and once in patient position, which is not allowed in a linear system. It is clear that at least for the semantic component, linearity is clearly too strict a requirement.

The reason for using an intuitionistic system rather than a linear system is that I wanted to analyse unspecified reflexive objects while keeping the grammatical system as simple as possible. I believe that this simplification is justified by the fact that the linearity restrictions ACG can be relaxed to some extend, while preserving the desirable features of the linear system. [Salvati, 2010, Kanazawa, 2007].

[7] I want to thank prof. Moortgat for pointing out the importance of linearity in ACG.

3.8 Optionality and Compositionality

Informally speaking, signs describe pieces of language. The morpho-syntactic component of a sign of a word describes how that word is written or spoken, the semantic component describes its meaning. Other components could be added to describe the prosody, distribution, and other aspects of a word. Signs can be combine using rules to form compound signs[8], and the ways in which signs combine model the way pieces of language can combine.

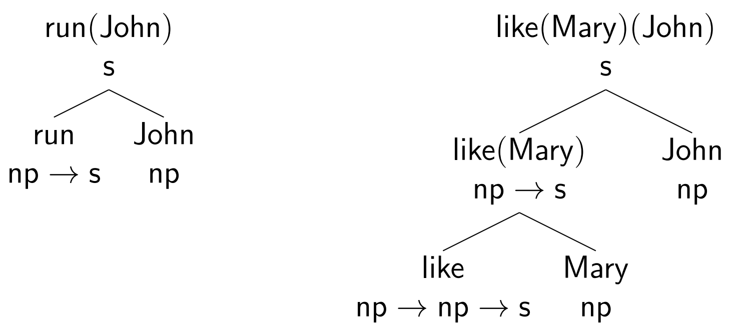

Figure 4: Application

In addition to the standard ways to combine words in categorial grammar and Montague grammar of *application* and *abstraction*, there is one new way to combine pieces of language: *option injection*, and a special piece of language *the null option* that make a compositional treatment of optionality possible. The null option and option injection are always available in the grammar, similar to application and abstraction. Using option types implicit arguments can be modeled.

Any term can have its optional arguments saturated with an option injected term of the appropriate type, if such a term is available. If no suitable term is available to serve as the argument, it can also be saturated with the null option:

3.9 Understanding Option Types

Since option types play such an important role in the proposed treatement of optionality, it is important that it is clear how option types work and how they can be used. In this section I will show how option types are used and provide some background information.

[8] Saussure calls a compound sign a *syntagm*

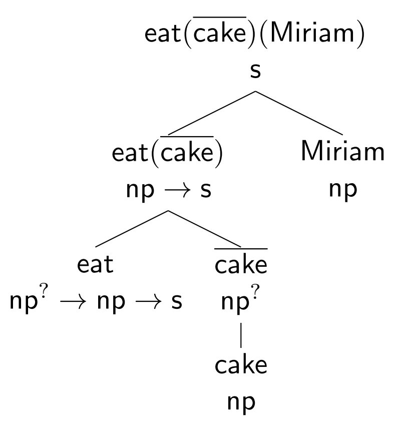

Figure 5: Application and option injection

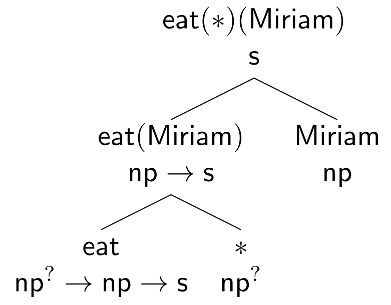

Figure 6: Application and ∗

To understand option types, one has to know what the values of an optional type can be. Suppose there is some term of type $a^?$: all we know is that it is optional. This term can have two kinds of content: either it is ordinary value with a marker that indicates it is provided, or a special term signifying the content is missing. An encapsulated provided term containing x is written as $\overline{x}$ (*option injection*), and the special 'missing content' term is written as $*$ (null option).

Option types are used in functional programming to emulate partial functions: functions that do not have a well-defined output for some inputs. As a proper function must always return something, some way of signalling that there is no real output is needed. This is done by making the result optional. If there is a result, the result is simply returned, encapsulated with an overline to signify that it is of an optional type. If there is no result, $*$ is returned instead. Functions with optional arguments allow one to emulate partial functions of type $a \to b$ as total functions of type $a \to b^?$. To exemplify this, suppose there is function named *lookup*, that takes a key of type $x : k$ and a collection of key-value pairs $k \times v$ and returns the value y with the given key. If there is such a key in the collection, the corresponding value $\overline{v}$ is returned, but if no such key is present $*$ is returned instead. The function *lookup* therefore has type $k \to (k \times v) \to v^?$. By using option types for return types, functions that do not have proper outputs for all inputs can be defined.

In the treatment of optional arguments, option types are used in a different way. Here they are not used to model functions with possibly missing results (optionality of output) but to define functions with possibly missing arguments (optionality of input) [9]. Instead of thinking of $*$ as a 'missing' output, think of $*$ as a 'provided' input. Since $*$ is always available in the grammar, any function with optional argument slots can have them saturated with $*$. $*$ has no internal content, the only information it carries is that no real content is available, which is why it is called the *null option*

To safely use the values contained in an optional value, a form of case analysis is needed. There are two cases, a value of optional type can be either 'present' ($\overline{x}$) or 'missing' ($*$). Any function that takes an optional value must deal with the missing case, as $*$ cannot provide any meaningful information. Accessing the content of an option type therefore requires that we specify how both cases are handled, this is done with the option operator.

The option operator is used to safely inspect the information of optional values, by applying a function if the argument is provided or providing a default value if it is missing. It takes an optional value $o : A^?$, which can be either a 'provided' value $\overline{x}$ or a 'missing' value $*$, a function $f : A \to B$ and a default value $d : B$. This implies that option is of type $(A^? \times (A \to B) \times B) \to B$.
If o is a provided value (i.e. of form $\overline{x}$), then f is applied to x, resulting in $f(x) : B$.
If o is $*$, the function f can not be applied, so instead the default $d : B$ is returned instead. In both cases, a value of type B is returned.

3.10 How to deal with optional values

To get a better feel for how option types can be used I will give some examples. Suppose we have a function *eats* that models the morpho-syntactic behavior of the verb *eat*. *eat* which takes an optional string o, a string s and returns the concatenation of s to the string `eats` o if

[9]This is just a difference in how option types are used, the definitions will also work for optional results

present.

$$eat : f^? ff = \lambda o_{f^?}.\lambda s_f.s \bullet \texttt{eats} \bullet \mathsf{option}(o_{f^?}, (\lambda x.x)_{f \to f}, \epsilon_f)$$

Since o is an optional argument, it is necessary to specify how the optional argument is handled. This is done using option, of which the resulting value depends on whether the argument is provided or omitted. If the argument is $*$ it returns the default argument, which is ϵ, the empty string. If the argument is $\overline{x}$, the extraction function is applied to x, which is the identity function in this case.

The function *eats* can not directly take ordinary values as arguments as the types do not match. However, any ordinary value can be 'optionalized' into a 'provided' optional value using the option injection rule, so that it can serve as an argument to *eats*
Since option injection is a rule of the ACG system it is always available for any term. With option injection it is possible to encapsulate an ordinary term so that it can be used as an argument to a function with an optional argument.

When there is no argument available to fill the optional argument slot of $f : a^? \to b$, it is still possible to get the result of f. In this case f is applied to the 'universal slot filler' : the null option $*$. The special $* : a^?$ term can always be introduced for any type a with the *universal filler* rule. This means that $*$ can be used as an argument to any function that requires an optional argument.

Below is an example of applying $*$ and an ordinary argument to `eats` :

$eat(*)(\texttt{Miriam}) =$
$(\lambda o.\lambda s_f.s \bullet \texttt{eats} \bullet \mathsf{option}(o, \lambda x.x, \epsilon))(*)(\texttt{Miriam}) =$
beta reduction, substitute $*$ for o and substitute `Miriam` for s
$\texttt{Miriam} \bullet \texttt{eats} \bullet \mathsf{option}(*, \lambda x.x, \epsilon) =$
option elimination, the case is $*$, the default value ϵ is returned
$\texttt{Miriam} \bullet \texttt{eats} \bullet \epsilon =$
ϵ is the empty string, so it can be omitted
$\texttt{Miriam} \bullet \texttt{eats}$

eats can also be supplied with a provided optional argument, like $\overline{\texttt{cake}}$. Here `cake` is encapsulated by option injection to $\overline{\texttt{cake}}$, so that it can serve as an argument to *eats*. The identity function is then applied to it with the option operator, to extract the content (`cake`), which is then concatenated.

$eat(\overline{\texttt{cake}})(\texttt{Miriam}) =$

$(\lambda o.\lambda s_f.s \bullet \textsf{eats} \bullet \textsf{option}(o, \lambda x.x, \epsilon))(\overline{\texttt{cake}})(\texttt{Miriam}) =$

beta reduction, substitute $\overline{\texttt{cake}}$ for o and substitute `Miriam` for s

$\texttt{Miriam} \bullet \textsf{eats} \bullet \textsf{option}(\overline{\textsf{cake}}, \lambda x.x, \epsilon) =$

option elimination, the case is $\overline{\texttt{cake}}$, so `cake` is applied to $\lambda x.x$

$\texttt{Miriam} \bullet \textsf{eats} \bullet (\lambda x.x)(\texttt{cake}) =$

beta reduction, replace x by `cake`

$\texttt{Miriam} \bullet \textsf{eats} \bullet \textsf{cake}$

As will become clear in the following chapters, option types are an essential extension to treat optional arguments in ACG: they allows us to specify arguments that may be omitted, and how to deal with omitted values. Note that the definition of the option types is very general, and can be used for things other than optional arguments. The definition used here is the result of taking the standard definitions of unit and sum types, using these to define option types and simplifying these as much as possible. Because of this there are many ways to use option types in ways that do not make sense from a linguistic point of view, or have nothing do with optional arguments.

For example, it is possible to use option types to define functions with optional results. This capability is not relevant for the treatment of optional arguments. The definition of option types is much more general than what is really needed to treat implicit arguments. However, I do not want +extend ACG with a highly constrained definition of option types because ACG is a very general framework and constraining the definition of options type to treat a single kind of optional argument would prevent the analysis of other phenomena related to optional arguments. The bottom line is that option types provide a general way of working with optional values, but say very little about what optional argument are or are not. In the following chapter I will show how option types can be used in a more restrictive manner, based on the procedure in [Blom et al., 2012], that only allow optionality in ways that can be motivated from a linguistic perspective.

3.11 Events in ACG

For the semantic component of the grammar a neo-Davidsonian event semantics is used. This particular flavor of model-theoretic semantics is used because of the greater flexibility it provides over standard Montagovian frameworks. In standard Montagovian frameworks the coupling between the argument structure of the syntactic and semantic components is very tight. A word that requires syntactic n arguments must have a denotation that takes n arguments, and the order of the arguments determines the role in the semantics. In neo-Davidsonian semantics this coupling between argument structure and semantic roles is much looser: it is possible to vary the semantic roles involved, independently of the provided syntactic arguments. This loose coupling of syntactic and semantic argument structure is exactly what is needed to treat certain types of implicit arguments where the semantic component may require more or fewer arguments than the syntactic component. A second reason to use event semantics is for comparison with [Winter and Zwarts, 2011], since this paper on event semantics in ACG introduces some of the problems I try to solve here.

3.11.1 Theta Roles

In a neo-Davidsonian framework a verb provided with all its arguments denotes a set of events. In the abstract component its type is vp, which surfaces as a string f and denotes a set of events et. In order to define the relation between entities and events, each argument of a verb is associated with a θ-role. A θ-role specifies in what way an entity was involved in an event. The most important roles are the agent (who performed the event) the patient (who undergoes the event) and goal (who is the recipient of the event). In a normal active sentence, the subject of a verb denotes the agent, the direct object denotes the patient and the second object denotes the goal. To use θ-roles in the semantic component, special θ-role predicates are used, which are simply 2-place predicates that denote a relation between events and entities :

Table 1: Thematic role predicates

Role:	Predicate :	Truth conditions: true iff
agent	$\mathsf{AG}(\mathsf{e},a)$	a is the agent of event e
patient	$\mathsf{PAT}(\mathsf{e},p)$	p is the patient of event e
goal	$\mathsf{GOAL}(\mathsf{e},g)$	g is the goal of event e

A verb saturated with all its argument (vp) denotes a set of events (et). Because sentences denote truth-values, a verb saturated with all its arguments must have its event variable existentially closed. This closure operation is defined as the sign:

$$\begin{array}{l} \mathsf{EC} : \mathsf{vp} \to \mathsf{s} = \\ \langle\ \lambda x.x \quad\quad\ :\ ff \\ ,\ \lambda g.\exists \mathsf{e}.g(\mathsf{e}) :\ (et)t\ \rangle \end{array}$$

In the semantic component EC takes a set of events f and applies existential quantification to f. EC surfaces as the identity function over strings, in other words: EC does not affect the pheno grammar.

In neo-Davidsonian event semantics it is possible to add or remove roles, which is very useful for the treatment of implicit arguments, as demonstrated in the sentences below:

(21) a. John sinks the ship
$\exists \mathsf{e}.\mathbf{sink}_{et}(\mathsf{e}) \wedge \mathsf{PAT}(e, \mathbf{john}_e) \wedge \mathsf{AG}(e, \mathbf{the\_ship}_e)$
b. The ship sinks
$\exists \mathsf{e}.\mathbf{sink}_{et}(\mathsf{e}) \wedge \mathsf{PAT}(e, \mathbf{the\_ship}_e)$

In this example there are two uses of *sink*, a transitive and an intransitive use. In the transitive case, there is an event predicate $\mathbf{sink}_{et}$, an agent and a patient, while in the intransitive case there is no agent. In both cases, the same event predicate is used, which explains the relations between (21a) and (21b), without resorting to meaning postulates.

The advantage of using a neo-Davidsonian semantics is that we do not need to resort to meaning postulates to define the relations between the different forms of a verb. Consider the verb *sink*, which has a transitive form with 2 syntactic argument (object and subject) and 2 semantic arguments (agent and patient), and an intransitive form with 1 syntactic (subject) and 1 semantic argument (patient). In a Montagovian framework we would require 2 distinct predicates for each form. It would certainly be possible to select the appropriate entry based on the presence of the optional argument using the option operator, but even so, meaning postulates would be required to relate the denotations of each verb form. In a neo-Davidsonian framework, only one predicate is required for all the forms of a single verb (the event predicate), other arguments be added trough theta-roles. In this framework of ACG and neo-Davidsonian semantics, the presence of an optional argument may only affect the semantic role associated with the optional argument. As the semantic roles are added by conjunction, the meanings of the verb forms are based on the verb predicate, plus the semantic roles involved. The relations between the different verb forms follow from the fact that the event predicate is shared by the different forms of a verb. MP's are not required to relate the different *forms* of a verb, however, they are still needed to describe relation between different verbs.

4 Signs for Verbs With and Without Optional Arguments

In this section I will explain how signs for ordinary verbs and verbs with optional argument such as UO's are specified in ACG with option types. First I will explain how the signs for verbs without optional arguments are defined, and how the abstract type and argument structure are related. Once this is clear, I will show how option types can be used to define signs of verbs that allow optional arguments, and give examples for the verb classes discussed in 2.3. Finally I will show how to derive verbs with option arguments from verbs without optional arguments.

4.1 Verbs with Obligatory Arguments

Below are definitions and explanations of how verbs with only obligatory arguments are defined in ACG. The signs for transitive and intransitive verbs are based on the usual treatments of verbs in Montague grammar and categorial grammar. An important observation is that the *type* of a sign determines the number or arguments it takes. Intransitive verbs take one, transitive verbs take two, and ditransitive take three. The number of arguments are reflected in the number of arguments in the abstract type of the sign.

4.1.1 Transitive Verbs

Where intransitive verbs only take a subject argument, transitive verbs also require an object argument. Often these two arguments are both noun phrases, but other types are possible as well. A typical transitive verb has abstract type $\mathsf{np} \to \mathsf{np} \to \mathsf{vp}$, semantic type *eeet* and syntactic type *fff*. Thus, a transitive verb requires two noun phrases to form a vp, it denotes a relation between two entities and events and surfaces as a function from two strings to a string. Below is an example of the sign of the transitive verb *build*:

$$\begin{array}{lll} \multicolumn{3}{l}{\mathsf{LIKES} : \mathsf{np} \to \mathsf{np} \to \mathsf{vp} =} \\ \langle & \lambda o.\lambda s.s \bullet \texttt{likes} \bullet o & : \ fff \\ , & \lambda p.\lambda a.\lambda \mathsf{e}.\mathbf{likes}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, a) \wedge \mathsf{PAT}(\mathsf{e}, p) & : \ eeet \ \rangle \end{array}$$

The morpho-syntactic component of LIKE is a function which takes a string s, corresponding to the subject, and a string o corresponding to the object, and returns the concatenation of s to the string `likes` and then to o. The semantic component is a function that takes an entity p, an entity a and an event e, and asserts that e must be a building event, that a must be the agent of e and that p must be the patient of e in order for the sentence to be true.

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{EC(LIKES(MARY)(JOHN)) : s} =} \\
\langle\ \texttt{John} \bullet \texttt{likes} \bullet \texttt{Mary} & :\ f \\
,\ \exists \mathsf{e}.\mathbf{likes}_{et}(\mathsf{e}) \land \mathsf{AG}(\mathsf{e}, \mathbf{john}_e) \land \mathsf{PAT}(\mathsf{e}, \mathbf{mary}_e) & :\ t\ \rangle
\end{array}$$

4.1.2 Ditransitive Verbs

Compared to transitive verb, ditransitive verbs like *give*, take an additional np argument. Because a ditransitive verb takes three np arguments, its abstract type is $\mathsf{np} \to \mathsf{np} \to \mathsf{np} \to \mathsf{vp}$, which implies that these surface as functions taking 3 strings, one from the direct object, one from the indirect object and one from the subject, to a string. A ditransitive verbs denotes a 4-place relation between 3 entities and an event. Below is an example of the sign of the transitive verb *give*:

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{GIVE : np \to np \to np \to vp} =} \\
\langle\ \lambda o.\lambda i.\lambda s.s \bullet \texttt{give} \bullet i \bullet o & :\ ffff \\
,\ \lambda p.\lambda g.\lambda a.\lambda \mathsf{e}.\mathbf{give}_{et}(\mathsf{e}) \land \mathsf{AG}(\mathsf{e}, a) \land \mathsf{PAT}(\mathsf{e}, p) \land GOAL(\mathsf{e}, g) & :\ eeeet\ \rangle
\end{array}$$

The semantic and morpho-syntactic components are similar to those of signs for transitive verbs, with the addition of an extra np argument. This extra argument is concatenated at the end in the morpho-syntactic component, and denotes the goal of the event. Below is the step-by-step construction of a sentence with a ditransitive verb:

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{GIVE(MARY) : np \to np \to vp} =} \\
\langle\ \lambda i.\lambda s.s \bullet \texttt{give} \bullet \texttt{Mary} \bullet i & :\ fff \\
,\ \lambda g.\lambda a.\lambda \mathsf{e}.\mathbf{give}_{et}(\mathsf{e}) \land \mathsf{PAT}(\mathsf{e}, \mathbf{mary}_e) \land \mathsf{AG}(\mathsf{e}, a) \land \mathsf{GOAL}(\mathsf{e}, g) & :\ eeet\ \rangle \\
\multicolumn{2}{l}{\mathsf{GIVE(MARY)(THESHIP) : np \to vp} =} \\
\langle\ \lambda s.s \bullet \texttt{give} \bullet \texttt{Mary} \bullet \texttt{the ship} & :\ ff \\
,\ \lambda a.\lambda \mathsf{e}.\mathbf{give}_{et}(\mathsf{e}) \land \mathsf{PAT}(\mathsf{e}, \mathbf{mary}_e) \land \mathsf{AG}(\mathsf{e}, a) \land \mathsf{GOAL}(\mathsf{e}, \mathbf{ship}_e) & :\ eet\ \rangle \\
\multicolumn{2}{l}{\mathsf{GIVE(MARY)(THESHIP)(JOHN) : vp} =} \\
\langle\ \texttt{John} \bullet \texttt{give} \bullet \texttt{Mary} \bullet \texttt{the ship} & :\ f \\
,\ \lambda \mathsf{e}.\mathbf{give}_{et}(\mathsf{e}) \land \mathsf{PAT}(\mathsf{e}, \mathbf{mary}_e) \land \mathsf{AG}(\mathsf{e}, \mathbf{john}_e) \land \mathsf{GOAL}(\mathsf{e}, \mathbf{ship}_e) & :\ et\ \rangle
\end{array}$$

4.2 Abstract Types and Argument Structure

In the signs for intransitive, transitive and ditransitive verbs there is a clear relation between the number of arguments (its *valency*), the kind of arguments the verb takes and the abstract type. Listing all the relevant information in one place makes this relation really obvious:

Verb class	Valency	Abstract type:	Examples:
intransitive	1	$\mathsf{np} \to \mathsf{vp}$	*run,fall*
transitive	2	$\mathsf{np} \to \mathsf{np} \to \mathsf{vp}$	*build, hit*
ditransitive	4	$\mathsf{np} \to \mathsf{np} \to \mathsf{np} \to \mathsf{vp}$	*give*
transitive/ditransitive	2 or 3	?	*introduce*
intransitive/transitive	1 or 2	?	*read, eat*

Table 2: Verb classes with a fixed number of arguments and their abstract types

In this table we can see that if a verb takes n arguments, then the abstract type of its sign also takes n arguments.

So far I have only described signs of verbs that take a fixed number of arguments, but some verbs are flexible in the number of arguments they take. Verbs like *read* or eat can be used as intransitive (1 argument) and as transitive verbs (2 arguments). This behavior cannot be described using only constants, functions and variables. To describe this flexibility in the argument structure, we need to specify *which* of the arguments are optional. This specification is done using *option types*, and is done by decorating the type of the optional argument with a question mark.

To model signs with varying numbers of arguments, option types are used to indicate that an argument is optional. Similar to how the number and types of arguments a verb takes determines the abstract type of its sign, the optionality of arguments determines which argument is optional. An optional argument can always be filled with $*$ when no suitable argument is present, or filled with a provided argument when it is. The fact that an optional argument (type $a^?$) is of a different type than the argument that is available for application (type a) is not an issue: any non-optional term $x : a$ can always be made into a provided optional term $\overline{x} : a^?$ using option injection. Thus, a sign with optional arguments can behave as if it requires the optional argument, or does not have the argument at all. For example: a sign for a transitive verb with an optional object (type $\mathsf{np}^? \to \mathsf{np} \to \mathsf{vp}$) can behave both as an intransitive verb (type $\mathsf{np} \to \mathsf{vp}$) or a transitive verb (type $\mathsf{np} \to \mathsf{np} \to \mathsf{vp}$).

There are multiple advantages to using optional arguments to define signs of verbs allowing UO or other types of optional arguments. The first is that since the specific forms of an ambitransitive verb are derived from a single lexical entry, the specific forms can be semantically related, without resorting to meaning postulates to establish the relations between the semantic components of the different forms. The second advantage is that deriving the specific forms requires no extra additions to the grammatical system, other than option types. The forms of verbs with a fixed number of arguments can be derived from a single ambitransitive form, using only the ACG system extended with option types. In chapter 6 I will explain this in detail.

In ACG the abstract type determines the concrete semantic and morpho-syntactic types. If the abstract type specifies that an argument is optional at the abstract level, these arguments are optional at the concrete levels. When such a concrete optional argument is provided, composition could go on as usual, but when it is missing some way of dealing with the missing information is required: it is the option operator. Using the option operator it can be specified how a missing argument should be handled. Each component can handle a missing argument in its own way. Because of this, it is possible to define optional arguments that when missing are not realized (they surface as the null string ϵ) but may still affect the semantics of the verb. In the next sections I will demonstrate how this can be used to define different kind of verbs that allow optional arguments.

4.3 Verbs with Optional Arguments

Now that I have explained how optionality is used to specify which arguments of verbs are optional (by marking the types of arguments with $\cdot^?$), I will explain how optional argument are dealt with in the concrete components, and how this can be used to model the verbs of the classes discussed in 2.3.

4.3.1 Verbs with Unspecified Objects

To demonstrate how signs for verbs with unspecified objects can be defined in ACG extended with option types, I give an example of the sign for the ambitransitive verb *read*. *read* has an intransitive and a transitive form, where the intransitive form has an existentially quantified patient. The transitive and intransitive forms can be thought of as specific instantiations of a single verb with one obligatory and one optional argument. We can use this insight to define a sign for *read*,

READ, which takes one optional $\mathsf{np}^?$ (the optional object), an np (the subject) and returns a vp.

$$
\begin{array}{l}
\mathsf{READ} : \mathsf{np}^? \to \mathsf{np} \to \mathsf{vp} = \\
\langle\ \lambda o.\lambda s.s \bullet \mathtt{read} \bullet \mathsf{option}(o, \lambda o'.o', \epsilon)\ :\ f^? f f \\
,\ \lambda p.\lambda a.\lambda \mathsf{e}.\ \mathbf{read}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, a) \wedge \\
\qquad \mathsf{option}(p\ , \lambda p'.\mathsf{PAT}(\mathsf{e}, p') \\
\qquad\qquad , \exists p'.\mathsf{PAT}(\mathsf{e}, p'))\ :\ e^? eet\ \rangle
\end{array}
$$

Because the object is optional, it is necessary to specify how to deal with a missing or provided argument in the semantic and morph-syntactic components. In the morpho-syntactic, when the object is provided, the object string is placed after the verb. If the object is missing, a null string is placed after the verb. In the semantic component, if the object is provided, the entity it denotes is used as the patient. If the object is missing, a patient is existentially introduced. By using this case analysis, all semantic arguments will be saturated, even if an argument is missing: there may be two semantic arguments for only one syntactic argument, or two semantic arguments for two syntactic arguments.

The sign READ can now compose with either two np arguments (an object and a subject) or a single np (an object). These two signs can serve as arguments to READ :

$$
\begin{array}{ll}
\mathsf{JOHN} : \mathsf{np} = & \mathsf{DUNE} : \mathsf{np} = \\
\langle\ \mathtt{John}\ :\ f & \langle\ \mathtt{Dune}\ :\ f \\
,\ \mathbf{john}_e\ :\ e\ \rangle & ,\ \mathbf{dune}_e\ :\ e\ \rangle
\end{array}
$$

Suppose that we know that these two signs provided as arguments to READ: JOHN as the subject, and DUNE as the object. If we want to use DUNE (type np) as the object of READ (type $\mathsf{np}^?$) by means of application, there is a problem, as np and $\mathsf{np}^?$ are not of the same type. Because of this mismatch of the types, function application cannot be used directly to set DUNE as the object. First DUNE must be made into a value of type $\mathsf{np}^?$. This is done using the option injection rule, which is always available. The option injection rule basically takes any ordinary value, and makes it into a *provided* optional argument. Applying option injection is notated by placing an overline over the injected value. The result of applying option injection to DUNE is:

$$
\begin{array}{l}
\overline{\mathsf{DUNE}} : \mathsf{np}^? = \\
\langle\ \overline{\mathtt{Dune}}\ :\ f^? \\
,\ \overline{\mathbf{dune}_e}\ :\ e^?\ \rangle
\end{array}
$$

Now that there is a value of type $\mathsf{np}^?$, it can be combined with READ using function application:
In the morpho-syntactic component, $\overline{\texttt{Dune}}$ is the object argument to the morpho-syntactic component of READ. In the semantic component, $\overline{\mathbf{dune}_e}$ is the patient argument to the semantic component of READ.

$$\begin{aligned}
&(\mathsf{READ} : \mathsf{np}^? \to \mathsf{np} \to \mathsf{vp})(\overline{\mathsf{DUNE}} : \mathsf{np}^?) = \\
&\langle\ (\lambda o_{f^?}.\lambda s_f.s \bullet \texttt{read} \bullet \mathsf{option}(o, \lambda o'.o', \epsilon))(\overline{\texttt{DUNE}} : f^?) \\
&,\ (\lambda p_{e^?}.\lambda a_e.\lambda \mathsf{e}_e.\mathbf{read}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, a) \wedge \\
&\quad \mathsf{option}(p, \lambda p'.\mathsf{PAT}(\mathsf{e}, p'), \exists p'.\mathsf{PAT}(\mathsf{e}, p')))(\overline{\mathbf{dune}_e : e^?})\rangle
\end{aligned}$$

After beta-reduction, we see that the $\overline{\texttt{Dune}}$ and $\overline{\mathbf{dune}_e}$ are analyzed by the option operator:

$$\begin{aligned}
&\mathsf{READ}(\overline{\mathsf{DUNE}}) : \mathsf{np} \to \mathsf{vp} = \\
&\langle\ \lambda s_f.s \bullet \texttt{read} \bullet \mathsf{option}(\overline{\texttt{DUNE}}, \lambda o'.o', \epsilon) \\
&,\ (\lambda p_{e^?}.\lambda a_e.\lambda \mathsf{e}_e.\mathbf{read}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, a) \wedge \\
&\quad \mathsf{option}(\overline{\mathbf{dune}_e}, \lambda p'.\mathsf{PAT}(\mathsf{e}, p'), \exists p'.\mathsf{PAT}(\mathsf{e}, p'))))\rangle
\end{aligned}$$

After option reduction, $\texttt{Dune}$ and $\mathbf{dune}_e$ are passed to the extraction functions. In the morpho-syntactic component, this is just the identity functions, and in the semantic component it is a function that assigns the patient role to its argument:

$$\begin{aligned}
&\mathsf{READ}(\overline{\mathsf{DUNE}}) : \mathsf{np} \to \mathsf{vp} = \\
&\langle\ \lambda s_f.s \bullet \texttt{read} \bullet (\lambda o'.o')(\texttt{DUNE}) \\
&,\ (\lambda p_{e^?}.\lambda a_e.\lambda \mathsf{e}_e.\mathbf{read}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, a) \wedge \\
&\quad (\lambda p'.\mathsf{PAT}(\mathsf{e}, p'))(\mathbf{dune}_e)\rangle
\end{aligned}$$

A final beta-reduction gives us the result of the application of the extraction functions:

$$\begin{aligned}
&\mathsf{READ}(\overline{\mathsf{DUNE}}) : \mathsf{np} \to \mathsf{vp} = \\
&\langle\ \lambda s.s \bullet \texttt{read} \bullet \texttt{Dune} && :\ ff \\
&,\ \lambda a.\lambda \mathsf{e}.\mathbf{read}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, a) \wedge \mathsf{PAT}(\mathsf{e}, \mathbf{dune}_e) && :\ eet\ \rangle
\end{aligned}$$

Then, the subject can be provided, by applying this to JOHN

$$\begin{aligned}
&\mathsf{READ}(\overline{\mathsf{DUNE}})(\mathsf{JOHN}) : \mathsf{vp} = \\
&\langle\ \texttt{John} \bullet \texttt{read} \bullet \texttt{Dune} && :\ f \\
&,\ \lambda \mathsf{e}.\mathbf{read}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, \mathbf{john}_e) \wedge \mathsf{PAT}(\mathsf{e}, \mathbf{dune}_e) && :\ et\ \rangle
\end{aligned}$$

Now that all arguments are provided, the event variable can be closed, yielding the sign for the sentence John read Dune:

$\mathsf{EC(READ(\overline{DUNE})(JOHN)) : s} =$
$\langle$ `John • read • Dune` $: f$
$, \exists \mathsf{e}.\mathbf{read}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, \mathbf{john}_e) \wedge \mathsf{PAT}(\mathsf{e}, \mathbf{dune}_e) : t \rangle$

If there is only one np argument available to READ, the optional argument slot is filled with the universal filler $*$ (type $\mathsf{np}^?$). In effect, this cause READ to behave as an intransitive verb:

$\mathsf{READ}(*) : \mathsf{np} \rightarrow \mathsf{vp} =$
$\langle \lambda s.s \bullet \mathtt{read} \bullet \epsilon : ff$
$, \lambda a.\lambda \mathsf{e}.\mathbf{read}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, a) \wedge \exists p'.\mathsf{PAT}(\mathsf{e}, p') : eet \rangle$

Now JOHN can be used as the subject of the intransitive form of *read* :

$\mathsf{READ}(*)(\mathsf{JOHN}) : \mathsf{vp} =$
$\langle$ `John • read •` $\epsilon : f$
$, \lambda \mathsf{e}.\mathbf{read}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, \mathbf{john}_e) \wedge \exists p'.\mathsf{PAT}(\mathsf{e}, p') : et \rangle$

In this sentence, all the arguments are provided. The missing object is replaced with $*$. Since the verb is saturated with all its arguments, the event variable can be closed, yielding the sign for the sentence John read:

$\mathsf{EC(READ}(*)\mathsf{(JOHN)) : s} =$
$\langle$ `John • read •` $\epsilon : f$
$, \exists \mathsf{e}.\mathbf{read}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, \mathbf{john}_e) \wedge \exists p'.\mathsf{PAT}(\mathsf{e}, p') : t \rangle$

It is clear by inspecting the semantic components that the entailment from the transitive to the intransitive form of the verb is accounted for. To show that existential entailments between the transitive and intransitive form holds, an existential quantifier is needed in the object position of the transitive form. For this we need a sign for *something*:

$\mathsf{SOMETHING} : (\mathsf{np} \rightarrow \mathsf{s}) \rightarrow \mathsf{s} =$
$\langle \lambda f.f(\mathtt{something}) : (ff)f$
$, \lambda f.\exists x.f(x) : (et)t \rangle$

Using abstraction and option injection, *something* is placed in object position. Note that the quantifier of *someone* is outside the scope of the quantifier that closes the event variable:

$\mathsf{SOMETHING}((\lambda x.\mathsf{EC}(\mathsf{READ}(\overline{x})(\mathsf{JOHN})))) : \mathsf{s} =$
$\langle$ `John • read • something` $: f$
$, \exists x.\exists \mathsf{e}.\mathbf{read}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, \mathbf{john}_e) \wedge \mathsf{PAT}(\mathsf{e}, x) : t \rangle$

Inspection of the semantic components of the signs for these sentences confirms that the equivalence is accounted for. In fact the semantic components of the signs for John read and John read something are equivalent. The only difference is the exact position of the existential quantifier, but under the standard rules of predicate logic these terms are equivalent.

The quantifier introduced by a missing argument of a UO has narrow scope. In the previous examples there was no quantifier in subject position, so there were no scope effects. To show that a UO quantifier always has narrow scope we need to verify that (22a) is treated as equivalent to the object narrow scope (ONS) of (22b), but not to the object wide scope (OWS) reading of (22b). For this, we need an additional quantifier:

$$\begin{array}{lcl}\multicolumn{3}{l}{\mathsf{EVERYONE} : (\mathsf{np} \to \mathsf{s}) \to \mathsf{s} =} \\ \langle\ \lambda f.f(\mathtt{everyone}) & : & (ff)f \\ ,\ \lambda g.\forall y.g(y) & : & (et)t\ \rangle\end{array}$$

(22) a. Everyone eats
b. Everyone eats something

1. Sign for (22a) :
   $\mathsf{EVERYONE}((\lambda x.\mathsf{EC}(\mathsf{EAT}(*)(x)))) : \mathsf{s} =$
   $\langle\ \mathtt{everyone} \bullet \mathtt{eat} \bullet \epsilon \quad : \ f$
   $,\ \forall y.\exists \mathsf{e}.\mathbf{eat}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, y) \wedge \exists x.\mathsf{PAT}(\mathsf{e}, x) \ : \ t\ \rangle$
2. Sign for the ONS reading of (22b) :
   $\mathsf{EVERYONE}((\lambda s.\mathsf{SOMETHING}((\lambda o.\mathsf{EC}(\mathsf{EAT}(\overline{o})(s)))))) : \mathsf{s} =$
   $\langle\ \mathtt{everyone} \bullet \mathtt{eat} \bullet \mathtt{something} \quad : \ f$
   $,\ \forall y.\exists x.\exists \mathsf{e}.\mathbf{eat}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, y) \wedge \mathsf{PAT}(\mathsf{e}, x) \ : \ t\ \rangle$
3. Sign for the OWS reading of (22b) :
   $\mathsf{SOMETHING}((\lambda o.\mathsf{EVERYONE}((\lambda s.\mathsf{EC}(\mathsf{EAT}(\overline{o})(s)))))) : \mathsf{s} =$
   $\langle\ \mathtt{everyone} \bullet \mathtt{eat} \bullet \mathtt{something} \quad : \ f$
   $,\ \exists x.\forall y.\exists \mathsf{e}.\mathbf{eat}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, y) \wedge \mathsf{PAT}(\mathsf{e}, x) \ : \ t\ \rangle$

Inspection of the semantic components of (2.) and (3.) confirms that (22a) is indeed treated as equivalent to the object narrow scope reading of (22b), but not to the object widescope reading.

4.3.2 Passives

Subcategorized prepositional phrases of passives, like *by* phrases, can be treated much like the object of a verb with UO's. Just like with unspecified objects, *by* phrases are optional arguments. When these optional arguments are missing, their unfilled slots are also saturated existentially. A difference is that passives do not take optional np signs, but optional np_{by} signs instead. These signs are formed by modifying an np sign, by prepending the appropriate preposition:

$$\begin{array}{l} \mathsf{BY} : \mathsf{np} \to \mathsf{np_{by}} = \\ \langle\ \lambda x.\mathsf{by} \bullet x \ :\ ff \\ ,\ \lambda x.x \qquad :\ ee\ \rangle \end{array}$$

By phrases are formed using the BY sign: BY takes a np and returns a np_{by} (a *by* phrase). Its morpho-syntactic denotation is a function that takes a string x and returns the string formed by the concatenation of `by` to x. BY denotes the identity function, it does not affect the meaning. The motivation for defining *by* phrases this way is that they should function as arguments to verbs, but should only be licensed to serve as arguments of passives.[10]

$$\begin{array}{l} \mathsf{BUILDpass} : \mathsf{np^?_{by}} \to \mathsf{np} \to \mathsf{vp} = \\ \langle\ \lambda b.\lambda s.s \bullet \texttt{was-build} \bullet \mathsf{option}(b, \lambda x.x, \epsilon)\ :\ f^? ff \\ ,\ \lambda p.\lambda a.\lambda \mathsf{e}.\ \mathbf{build}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, a) \wedge \\ \qquad \mathsf{option}(p, \mathsf{PAT}(\mathsf{e}), \exists p'.\mathsf{PAT}(\mathsf{e}, p'))\ :\ e^? eet\ \rangle \end{array}$$

This sign takes an optional $\mathsf{np_{by}}$ (the *by* phrase) and an obligatory np (the subject), and returns a vp. In the morpho-syntactic component, the string of the subject, `was`, the verb string, the string of the *by* phrase are concatenated. If the *by* phrase is missing, an empty string takes its place. In the semantic component, the denotation of the subject is the patient. If the *by* phrase is provided, the entity it denotes is used as the agent, and if it is missing, an agent is existentially introduced.

The list of signs below demonstrate the existential implicit argument of passive *build*, these are the signs used for *John* and *the ship*:

$$\begin{array}{ll} \mathsf{JOHN} : \mathsf{np} = & \mathsf{THESHIP} : \mathsf{np} = \\ \langle\ \texttt{John}\ :\ f & \langle\ \texttt{the ship}\ :\ f \\ ,\ \mathbf{john}_e\ :\ e\ \rangle & ,\ \mathbf{ship}_e\ :\ e\ \rangle \end{array}$$

For comparison with an active sentences, this is the sign used for the active from of the verb build :

[10]Note that this restriction only pertains to *by* as arguments to passives, no claims are made about using *by* as an adjective, as in "The book by Frank Herbert".

$$\begin{aligned}
&\mathsf{BUILD} : \mathsf{np} \to \mathsf{np} \to \mathsf{vp} = \\
&\langle\ \lambda o.\lambda s.s \bullet \texttt{build} \bullet o && : \ fff \\
&,\ \lambda p.\lambda a.\lambda \mathsf{e}.\mathbf{build}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, a) \wedge \mathsf{PAT}(\mathsf{e}, p) && : \ eeet\ \rangle
\end{aligned}$$

I haven't closed the event variable to emphasize the scope of the existential quantifier that saturates the missing argument slot in (2), note that is has the narrowest possible scope.

1. $\mathsf{BUILD}(\mathsf{THESHIP})(\mathsf{JOHN}) : \mathsf{vp} =$
   $\langle\ \texttt{John} \bullet \texttt{build} \bullet \texttt{the ship} \quad : \ f$
   $,\ \lambda \mathsf{e}.\mathbf{build}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, \mathbf{john}_e) \wedge \mathsf{PAT}(\mathsf{e}, \mathbf{ship}_e) \ : \ et\ \rangle$
2. $\mathsf{BUILDpass}(*)(\mathsf{THESHIP}) : \mathsf{vp} =$
   $\langle\ \texttt{the ship} \bullet \texttt{was-build} \bullet \epsilon \quad : \ f$
   $,\ \lambda \mathsf{e}.\mathbf{build}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, \mathbf{ship}_e) \wedge \exists p'.\mathsf{PAT}(\mathsf{e}, p') \ : \ et\ \rangle$
3. $\mathsf{BUILDpass}(\overline{\mathsf{BY}(\mathsf{JOHN})})(\mathsf{THESHIP}) : \mathsf{vp} =$
   $\langle\ \texttt{the ship} \bullet \texttt{was-build} \bullet \texttt{by} \bullet \texttt{John} \quad : \ f$
   $,\ \lambda \mathsf{e}.\mathbf{build}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, \mathbf{ship}_e) \wedge \mathsf{PAT}(\mathsf{e}, \mathbf{john}_e) \ : \ et\ \rangle$

4.3.3 Understood Reflexive Objects

Signs for verbs that have a reflexive interpretation when the object is missing can also be defined using optional arguments. When the optional object argument is missing, the missing slot is resolved using an empty string and by asserting that the agent is also the patient. When the optional argument is provided, this type of verb functions as a transitive verb. An example of such a sign is *shave*.

$$\begin{aligned}
&\mathsf{SHAVE} : \mathsf{np}^? \to \mathsf{np} \to \mathsf{vp} = \\
&\langle\ \lambda o.\lambda s.s \bullet \texttt{shaves} \bullet \mathsf{option}(o, \lambda o'.o', \epsilon) \ : \ f^? ff \\
&,\ \lambda p.\lambda a.\lambda \mathsf{e}.\ \mathbf{shave}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, a) \wedge \\
&\qquad \mathsf{option}(p\ , \lambda p'.\mathsf{PAT}(\mathsf{e}, p'), \mathsf{PAT}(\mathsf{e}, a)) \ : \ e^? eet\ \rangle
\end{aligned}$$

SHAVE denotes a function that takes an optional entity p, an entity a, and an event e, and returns true if e is a shaving event, a is the agent of the shaving, and if p is provided then p is the patient of the shaving, but if p is missing then a is the patient (in addition to being the agent). SHAVE surfaces as a function that takes an object string o, a subject string s, and returns the concatenation of s to `shave` to o if it is provided and to ϵ if missing.

To show the equivalence between verbs with overt reflexives and covert reflexives introduced by missing arguments a sign for reflexives is needed, which models words like *himself, herself* and *itself.* As the framework has no support for gender agreement, I will just assume it surfaces as `himself`.

$$\begin{array}{l} \mathsf{SELF} : (\mathsf{np} \to \mathsf{np} \to \mathsf{vp}) \to \mathsf{np} \to \mathsf{vp} = \\ \langle\ \lambda v.\lambda s.v(\texttt{himself})(s)\ :\ (fff)ff \\ ,\ \lambda v.\lambda a.\lambda \mathsf{e}.v(a)(a)(\mathsf{e})\ \ :\ (eeet)eet\ \rangle \end{array}$$

The semantic component of SELF is very similar to the standard Montagovian treatment of reflexivization, with the addition that the event variable is passed on. It takes a 2-place predicate, and two arguments. The first is an entity, which normally comes from the subject phrase, the second is an event argument. The entity is then applied twice to the verb (as agent and as patient), and the event argument is passed on.
The morpho-syntactic component SELF is a function that takes a string function v_{fff} of a transitive verb and a subject string. It returns the application of v to the subject and the string `himself` [11]
To show that the sentence John shaves and John shaves himself are equivalent, we compare the semantic component of the signs for these sentences. In order to build a sign for the second sentence, we need to apply himself to shave. As himself takes an argument of is of type $\mathsf{np} \to \mathsf{np} \to \mathsf{vp}$, while shave is of type $\mathsf{np}^? \to \mathsf{np} \to \mathsf{vp}$, application is not directly possible. First we need to make the optional argument of shave obligatory using option injection and abstraction:

$$\begin{array}{l} (\lambda x.\mathsf{SHAVE}(\overline{x})) : \mathsf{np} \to \mathsf{np} \to \mathsf{vp} = \\ \langle\ \lambda x.\lambda s.s \bullet \texttt{shaves} \bullet x \qquad\qquad\qquad\qquad :\ fff \\ ,\ \lambda x.\lambda a.\lambda \mathsf{e}.\mathbf{shave}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e},a) \wedge \mathsf{PAT}(\mathsf{e},x)\ :\ eeet\ \rangle \end{array}$$

The result of this has the appropriate type, that of a transitive verb, so now self can be applied:

$$\begin{array}{l} \mathsf{SELF}((\lambda x.\mathsf{SHAVE}(\overline{x}))) : \mathsf{np} \to \mathsf{vp} = \\ \langle\ \lambda s.s \bullet \texttt{shaves} \bullet \texttt{himself} \qquad\qquad\qquad\qquad\qquad\qquad :\ ff \\ ,\ \lambda a.\lambda \mathsf{e}.\mathbf{shave}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e},a) \wedge \mathsf{option}(p\ ,\lambda p'.\mathsf{PAT}(\mathsf{e},p'),\mathsf{PAT}(\mathsf{e},a))\ :\ eet\ \rangle \end{array}$$

Comparing the signs for the sentences, it is obvious that their denotations are the same:

[11] In a more realistic grammar the inserted string would depend on the features of s. For example, if s would have the features *singular* and *female*, then `herself` would be inserted instead of `itself` See [de Groote and Maarek, 2007] for a treatment of inflection in ACG.

1. $\mathsf{EC}(\mathsf{SHAVE}(*)(\mathsf{JOHN})) : \mathsf{s} =$
   $\langle\ \texttt{John} \bullet \texttt{shaves} \bullet \epsilon \quad : \ f$
   $,\ \exists \mathsf{e}.\mathbf{shave}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, \mathbf{john}_e) \wedge \mathsf{PAT}(\mathsf{e}, \mathbf{john}_e) \ : \ t\ \rangle$
2. $\mathsf{EC}(\mathsf{SELF}((\lambda x.\mathsf{SHAVE}(\overline{x})))(\mathsf{JOHN})) : \mathsf{s} =$
   $\langle\ \texttt{John} \bullet \texttt{shaves} \bullet \texttt{himself} \quad : \ f$
   $,\ \exists \mathsf{e}.\mathbf{shave}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, \mathbf{john}_e) \wedge \mathsf{PAT}(\mathsf{e}, \mathbf{john}_e) \ : \ t\ \rangle$

The ambitransitive sign for *shave* can of course also take an object when an object is provided, in which case the patient is the object.

$\mathsf{EC}(\mathsf{SHAVE}(\overline{\mathsf{BOB}})(\mathsf{JOHN})) : \mathsf{s} =$
$\langle\ \texttt{John} \bullet \texttt{shaves} \bullet \texttt{Bob} \quad : \ f$
$,\ \exists \mathsf{e}.\mathbf{shave}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, \mathbf{john}_e) \wedge \mathsf{PAT}(\mathsf{e}, \mathbf{bob}_e) \ : \ t\ \rangle$

4.3.4 Verbs with Understood Objects

Verbs with understood objects are somewhat similar to verbs with reflexive objects: when the object is missing, the entity associated with the subject is used twice. In reflexive verbs, the subject entity is also used as the object entity if there is no object provided. In verbs with understood objects, some part or property of the subject entity is used as the object entity when no external object is provided. For example, if *blink* does not receive an object argument to use as patient, the eyes of the subject are used as the patient instead:

$$\begin{aligned}
&\mathsf{BLINK} : \mathsf{np}^? \to \mathsf{np} \to \mathsf{vp} = \\
&\langle\ \lambda o.\lambda s.s \bullet \texttt{blinks} \bullet \mathsf{option}(o, \lambda o'.o', \epsilon) \ : \ f^? f f \\
&,\ \lambda p.\lambda a.\lambda \mathsf{e}.\ \mathbf{blink}_{et}(\mathsf{e}) \\
&\qquad \wedge \mathsf{AG}(\mathsf{e}, a) \\
&\qquad \wedge \mathsf{option}(p\ ,\lambda p'.\mathsf{PAT}(\mathsf{e}, p') \\
&\qquad\qquad ,\mathsf{PAT}(\mathsf{e}, \mathbf{of}_{(et)ee}(\mathbf{eyes}_{et})(a))) \ : \ e^? eet \quad \rangle
\end{aligned}$$

1. $\mathsf{EC}(\mathsf{BLINK}(*)(\mathsf{JOHN})) : \mathsf{s} =$
   $\langle\ \texttt{John} \bullet \texttt{blinks} \bullet \epsilon \quad : \ f$
   $,\ \exists \mathsf{e}.\mathbf{blink}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, \mathbf{john}_e) \wedge \mathsf{PAT}(\mathsf{e}, \mathbf{of}_{(et)ee}(\mathbf{eyes}_{et})(\mathbf{john}_e)) \ : \ t\ \rangle$
2. $\mathsf{EC}(\mathsf{BLINK}(\overline{\mathsf{OF}(\mathsf{EYES})(\mathsf{JOHN})})(\mathsf{JOHN})) : \mathsf{s} =$
   $\langle\ \texttt{John} \bullet \texttt{blinks} \bullet \texttt{eyes} \bullet \texttt{of} \bullet \texttt{John} \quad : \ f$
   $,\ \exists \mathsf{e}.\mathbf{blink}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, \mathbf{john}_e) \wedge \mathsf{PAT}(\mathsf{e}, \mathbf{of}_{(et)ee}(\mathbf{eyes}_{et})(\mathbf{john}_e)) \ : \ t\ \rangle$

4.3.5 Unaccusatives: Optional Agent Arguments

Many unaccusative verbs are related to a causative transitive form. Since there are two forms, an unaccusative (intransitive) form and a

transitive form, with related meanings, it is possible to postulate a single form with an ordinary and an optional argument from which both forms can be derived.
For unaccusatives that have a transitive counter-part, a similar approach is taken as to UO's, but instead of saturating an unfilled argument slot using existential import, it is saturated with a neutral element.

When the argument is missing, it simply does not contribute anything to the meaning, and when the extra argument is provided, it is used as the agent. The presence also triggers a change in word order, the extra object takes the subject position, and the other argument, which would otherwise be the subject, takes the object position. The abstract type for these type of verbs is $\mathsf{np} \to \mathsf{np}^? \to \mathsf{s}$, because usually the agent argument corresponds with the subject argument (the second np), which is optional for unaccusatives that have a transitive counterpart. An example of an unaccusative that also has a transitive form is *break*. A sign for this verb that relates these forms is defined as:

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{BREAK} : \mathsf{np} \to \mathsf{np}^? \to \mathsf{vp} =} \\
\langle\ \lambda o.\lambda s.\mathsf{option}(s, \lambda s'.s' \bullet \texttt{broke} \bullet o, o \bullet \texttt{broke}) & :\ ff^?f \\
,\ \lambda p.\lambda a.\lambda \mathsf{e}.\mathbf{break}_{et}(\mathsf{e}) \wedge \mathsf{option}(a\ ,\lambda a'.\mathsf{AG}(\mathsf{e}, a'),\top) \wedge \mathsf{PAT}(\mathsf{e}, p) & :\ ee^?et\ \rangle
\end{array}$$

In the morpho-syntactic component BREAK takes an object string and an optional subject string. When the subject is present, the concatenation of the subject string, the verb string and the object string is returned, but when it is missing the concatenation of the object string and verb string is returned[12]. In the semantic component, the variable p supplies the patient. The optional variable s supplies the agent when present, but when absent, no agent is set.

Using this sign, entailments between unaccusatives and their transitive counterparts can be modeled: With this sign for *break*, we can give an analysis of the following entailments:

(1.) John broke the window.
$\Downarrow$
(2.) Someone broke the window.
$\Downarrow$
(3.) The window broke.

[12] Note that the so-called object string takes the position of the subject when the argument that supplies the agent is missing.

The signs for these sentences are:

$$\begin{array}{l}
\mathsf{EC(BREAK(THEWINDOW)(\overline{JOHN}))} : \mathsf{s} = \\
\langle \; \texttt{John} \bullet \texttt{broke} \bullet \texttt{the window} \qquad\qquad\qquad\quad : \; f \\
, \; \exists \mathsf{e}.\mathbf{break}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, \mathbf{john}_e) \wedge \mathsf{PAT}(\mathsf{e}, \mathbf{window}_e) \; : \; t \; \rangle
\end{array}$$

$$\begin{array}{l}
\mathsf{SOMEONE}((\lambda x.\mathsf{EC(BREAK(THEWINDOW)}(\overline{x})))) : \mathsf{s} = \\
\langle \; \texttt{someone} \bullet \texttt{broke} \bullet \texttt{the window} \qquad\qquad : \; f \\
, \; \exists x.\exists \mathsf{e}.\mathbf{break}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, x) \wedge \mathsf{PAT}(\mathsf{e}, \mathbf{window}_e) \; : \; t \; \rangle
\end{array}$$

$$\begin{array}{l}
\mathsf{EC(BREAK(THEWINDOW)}(*)) : \mathsf{s} = \\
\langle \; \texttt{the window} \bullet \texttt{broke} \qquad\qquad : \; f \\
, \; \exists \mathsf{e}.\mathbf{break}_{et}(\mathsf{e}) \wedge \top \wedge \mathsf{PAT}(\mathsf{e}, \mathbf{window}_e) \; : \; t \; \rangle
\end{array}$$

The semantic components show the desired entailments.

4.3.6 Overview

Now that we know what the types of the verb with optional arguments are, table 2 can be extended with verb classes with optional arguments:

Verb class	Valency	Abstract type	Examples
intransitive	1	$\mathsf{np} \to \mathsf{vp}$	*run,fall*
transitive	2	$\mathsf{np} \to \mathsf{np} \to \mathsf{vp}$	*build, hit*
transitive with optional object	1 or 2	$\mathsf{np}^? \to \mathsf{np} \to \mathsf{vp}$	*eat, sings*
unaccusative with optional agent	1 or 2	$\mathsf{np} \to \mathsf{np}^? \to \mathsf{vp}$	*sink, break*
passive transitive	1 or 2	$\mathsf{np}^?_{\mathsf{by}} \to \mathsf{np} \to \mathsf{vp}$	*was build*
ditransitive with opt. indirect obj.	2 or 3	$\mathsf{np} \to \mathsf{np}^? \to \mathsf{np} \to \mathsf{vp}$	*give, tell*
ditransitive with opt *to* phrase	2 or 3	$\mathsf{np} \to \mathsf{np}^?_{\mathsf{to}} \to \mathsf{np} \to \mathsf{vp}$	*give to, introduce*
passive ditransitive	1,2 or 3	$\mathsf{np}^?_{\mathsf{by}} \to \mathsf{np}^?_{\mathsf{to}} \to \mathsf{np} \to \mathsf{vp}$	*was given*

Table 3: Verb classes and their types

Note that the position of the option type marker already tells us a lot about how the optionality will affect the interpretation of the verb. For transitives with an optional object, something happens to the patient when missing, because the entity that saturates the patient argument normally comes from the object, which corresponds to the first np argument. The same applies to the indirect object of ditransitives. For unaccusatives, something related to the agent must happen, as the second np is marked optional, which normally supplies the agent. For the various passive constructions, it clear that the optionality affects the *by* phrases, as the arguments corresponding to these phrases are marked as optional.

4.4 Optionalizing Operators

Option types can be regarded too general for treating optional arguments: they can be used to define signs with optionality that have no obvious linguistic relevance. For example, nothing prevents the definition of a sign $\mathsf{strange}$ with type $\mathsf{np} \to \mathsf{s}?$. This $\mathsf{strange}$ sign is much like an intransitive verb, but its result may be missing: it might be realized as $*$ and denote $*$. This is clearly undesirable[13]. Another use of option types that is undesirable is letting the form and meaning depend too much on the presence of an argument. A reason to use optional arguments is to give concise descriptions of the relation between certain related verbs with different number of arguments. With the definition of option types as presented here, one can take things too far, and postulate verbs with optional arguments that have completely different meanings when provided with different numbers of arguments. For example, using option types we can define a sign *run-build*, which is the same as *run* if provided with one argument and the same as *build* if provided with two arguments.

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{RUNBUILD} : \mathsf{np}^? \to \mathsf{np} \to \mathsf{vp} =} \\
\langle\ \lambda o.\lambda s.\mathsf{option}(o, \lambda o'.s \bullet \mathtt{build} \bullet o', s \bullet \mathtt{run}) & :\ f^? ff \\
,\ \lambda p.\mathsf{option}(p\ ,\mathbf{build}_{eeet},\mathbf{run}_{eet}) & :\ e^? eet\ \rangle
\end{array}$$

To my knowledge there is no linguistic justification for signs, where the meaning and form of an ambitransitive verb are not related when the optional arguments are missing. Surely, such absurdities should be avoided. To avoid such absurdities, a structured method to introduce optional arguments is required. One way of doing this is to only allow the introduction of option types using *operators* that take ordinary signs and add optionality in a specific manner. These operators conveniently formalize rules that describe how a verb with optional arguments is derived from a verb with only obligatory arguments.

[13]In semantics, functions with optional results can be used to model partial functions, and optional results make sense in topics like partial logics. In syntax however, I can not really think of any reason to allow optional results.

4.4.1 Existential Optional Arguments

One such operator would be useful to introduce optionality to transitive verbs that allow existential optional arguments. The operator would simply take a transitive verb and return a verb with an optional object, which is interpreted as an understood existential argument when the object is missing. This operator can be defined as a sign:

$$\begin{array}{lcl} \mathsf{UO} : (\mathsf{np} \rightarrow \mathsf{np} \rightarrow \mathsf{vp}) \rightarrow \mathsf{np}^? \rightarrow \mathsf{np} \rightarrow \mathsf{vp} = & & \\ \langle\ \lambda v.\lambda o.\lambda s.v(\mathsf{option}(o, \lambda o'.o', \epsilon))(s) & : & (fff)f^?ff \\ ,\ \lambda v.\lambda o.\lambda s.\lambda \mathsf{e}.\mathsf{option}(o, \lambda o'.v(o')(s)(\mathsf{e}), \exists o'.v(o')(s)(\mathsf{e})) & : & (eeet)e^?eet\ \rangle \end{array}$$

UO takes a transitive verb, and returns a verb with an optional object and an obligatory subject. In the resulting ambitransitive verb, the missing argument slot is saturated with the empty string and existential quantification when the object is missing, and functions like transitive verb if the object is present

Suppose we have a sign for the transitive form of *read*:

$$\begin{array}{lcl} \mathsf{READtv} : \mathsf{np} \rightarrow \mathsf{np} \rightarrow \mathsf{vp} = & & \\ \langle\ \lambda o.\lambda s.s \bullet \mathtt{read} \bullet o & : & fff \\ ,\ \lambda p.\lambda a.\lambda \mathsf{e}.\mathbf{read}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, a) \wedge \mathsf{PAT}(\mathsf{e}, p) & : & eeet\ \rangle \end{array}$$

The object of READtv should be optional, interpreted as an understood existential object when missing, so UO must be applied to READtv. The result of this is equal to the optional version of READ:

$$\begin{array}{l} \mathsf{UO(READtv)} : \mathsf{np}^? \rightarrow \mathsf{np} \rightarrow \mathsf{vp} = \\ \langle\ \lambda o.\lambda s.s \bullet \mathtt{read} \bullet \mathsf{option}(o, \lambda o'.o', \epsilon)\ :\ f^?ff \\ ,\ \lambda o.\lambda s.\lambda \mathsf{e}.\mathsf{option}(o, \lambda o'.\mathbf{read}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, s) \wedge \mathsf{PAT}(\mathsf{e}, o') \\ \qquad\qquad\qquad\qquad , \exists o'.\mathbf{read}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, s) \wedge \mathsf{PAT}(\mathsf{e}, o'))\ :\ e^?eet\ \rangle \end{array}$$

Using the operator UO, it is possible to analyze the same sentences as in 4.3.1 using an optionalized transitive verb:

1. $\mathsf{EC(UO(READtv)(\overline{DUNE})(JOHN))} : \mathsf{s} =$
$$\begin{array}{lcl} \langle\ \mathtt{John} \bullet \mathtt{read} \bullet \mathtt{Dune} & : & f \\ ,\ \exists \mathsf{e}.\mathbf{read}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, \mathbf{john}_e) \wedge \mathsf{PAT}(\mathsf{e}, \mathbf{dune}_e) & : & t\ \rangle \end{array}$$

2. $\mathsf{EC(UO(READtv)(*)(JOHN))} : \mathsf{s} =$
$$\begin{array}{lcl} \langle\ \mathtt{John} \bullet \mathtt{read} \bullet \epsilon & : & f \\ ,\ \exists \mathsf{e}.\exists o'.\mathbf{read}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, \mathbf{john}_e) \wedge \mathsf{PAT}(\mathsf{e}, o') & : & t\ \rangle \end{array}$$

3. $\mathsf{SOMETHING}((\lambda x.\mathsf{EC}(\mathsf{UO}(\mathsf{READtv})(\overline{x})(\mathsf{JOHN})))) : \mathsf{s} =$
$\langle$ `John • read • something` $: f$
$, \exists x.\exists \mathsf{e}.\mathbf{read}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, \mathbf{john}_e) \wedge \mathsf{PAT}(\mathsf{e}, x) : t \rangle$

The UO operator is very similar to the Unspecified Object Deletion rule of Dowty [Dowty, 1982], but with a crucial difference: instead of removing the object argument, it optionalizes the object argument. So instead of a rule that deletes the object argument slot from a verb, we have a rule that makes the object slot optional, allowing removal at a later stage by the compositional system if needed.
Dowty uses a different framework, so it is impossible to directly compare his rule with mine, but it is possible to give a sign (UOD) that is very similar to his Unspecified Object Deletion rule:

$$\begin{aligned}&\mathsf{UOD} : (\mathsf{np} \to \mathsf{np} \to \mathsf{vp}) \to \mathsf{np} \to \mathsf{vp} =\\ &\langle\ \lambda v.\lambda s.v(\epsilon)(s) \quad : (fff)ff\\ &,\ \lambda v.\lambda s.\lambda \mathsf{e}.\exists o'.v(o')(s)(\mathsf{e}) : (eeet)eet\ \rangle\end{aligned}$$

UOD can be compared with UO: UOD is a specific instance of UO. When there is no suitable argument to fill the result of UO, it is filled with $*$ and the result is exactly the same as UOD. This is easy to verify, by checking that:

$$\lambda v_{\mathsf{np}\to\mathsf{np}\to\mathsf{vp}}.\mathsf{UO}(v)(*) = \mathsf{UOD}$$

These two are indeed equivalent:

$$\begin{aligned}&(\lambda v.\mathsf{UO}(v)(*)) : (\mathsf{np} \to \mathsf{np} \to \mathsf{vp}) \to \mathsf{np} \to \mathsf{vp} =\\ &\langle\ \lambda v.\lambda s.v(\epsilon)(s) \quad : (fff)ff\\ &,\ \lambda v.\lambda s.\lambda \mathsf{e}.\exists o'.v(o')(s)(\mathsf{e}) : (eeet)eet\ \rangle\end{aligned}$$

This equivalence shows that Dowty's rule is a specific instance of the option type approach to implicit arguments.

4.4.2 Passivization

Given the strict relation between the forms and meanings transitive verbs and their passive forms, it is not surprising that it is possible to define an operator that describes this relation. Below is the rule that maps a transitive verb to its passive form:

$$\begin{aligned}&\mathsf{PASStv} : (\mathsf{np} \to \mathsf{np} \to \mathsf{vp}) \to \mathsf{np}^{?}_{\mathsf{by}} \to \mathsf{np} \to \mathsf{vp} =\\ &\langle\ \lambda v.\lambda o.\lambda s.v(\mathsf{option}(o, \lambda o'.o', \epsilon))(s \bullet \texttt{was}) \quad : (fff)f^{?}ff\\ &,\ \lambda v.\lambda o.\lambda s.\lambda \mathsf{e}.\mathsf{option}(o, \lambda o'.v(s)(o')(\mathsf{e}), \exists o'.v(s)(o')(\mathsf{e})) : (eeet)e^{?}eet\ \rangle\end{aligned}$$

The PASS operator takes a transitive verb, an optional *by* phrase and a subject np. In the morpho-syntactic component it concatenates the subject with an auxiliary verb, the verb string, and the *by* phrase if it is present. In the semantic component it takes a 2-place predicate, an entity coming from the *by* phrase, an entity coming from the subject, and an event variable. The entities are all applied to the predicate in almost the same order, but the *by* phrase entity and subject entity switch places. When the entity coming from the *by* phrase is missing, the empty slot is existentially quantified over.

Note that PASStv and UO are very similar: compared to UO, PASS takes an optional *by* phrase instead of an optional noun phrase, switches the order of the arguments coming from the object and subject around in the semantic component, and adds an auxiliary verb, but its effect on optionality is almost similar.
For the passivization of ditransitive verbs a similar operator can be given:

$$\begin{array}{l}
\mathsf{PASSdtv} : (\mathsf{np} \to \mathsf{np}^?_{\mathsf{to}} \to \mathsf{np} \to \mathsf{vp}) \to \mathsf{np}^?_{\mathsf{by}} \to \mathsf{np}^?_{\mathsf{to}} \to \mathsf{np} \to \mathsf{vp} = \\
\langle\ \lambda v.\lambda b.\lambda t.\lambda s.v(\mathsf{option}(b, \lambda b'.b', \epsilon))(t)(s \bullet \mathtt{was}) \ : \ (ff^?ff)f^?f^?ff \\
,\ \lambda v.\lambda b.\lambda t.\lambda s.\lambda \mathsf{e}.\mathsf{option}(b\ , \lambda b'.v(s)(t)(b')(\mathsf{e}) \\
\qquad\qquad\qquad\qquad\qquad , \exists b'.v(s)(t)(b')(\mathsf{e})) : \ (ee^?eet)e^?e^?eet\ \rangle
\end{array}$$

The $\mathsf{PASSdtv}$ operator simply takes an additional optional argument compared to the PASStv operator. Because there are now two optional arguments, both optional arguments need to be inspected. The morpho-syntactic component does not become much more complex: the extra optional argument is filled using ϵ when missing, but in the semantic component a nested use of option is required to properly introduce the existential quantifiers for the two optional arguments.
Assume that we have this sign for the ditransitive verb *introduce* :

$$\begin{array}{l}
\mathsf{INTRODUCE} : \mathsf{np} \to \mathsf{np}^?_{\mathsf{to}} \to \mathsf{np} \to \mathsf{vp} = \\
\langle\ \lambda o.\lambda i.\lambda s.s \bullet \mathtt{introduced} \bullet o \bullet \mathsf{option}(i, \lambda i'.i, \epsilon) \ : \ ff^?ff \\
,\ \lambda p.\lambda g.\lambda a.\lambda \mathsf{e}.\ \mathbf{introduce}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, a) \wedge \mathsf{PAT}(\mathsf{e}, p) \\
\qquad\qquad \wedge \mathsf{option}(g\ , \lambda g'.\mathsf{GOAL}(\mathsf{e}, g'), \exists g'.\mathsf{GOAL}(\mathsf{e}, g')) \ : \ ee^?eet\ \rangle
\end{array}$$

Application of the passivization operator for ditransitive verbs to this sign gives the sign for passive *introduce*:

$$
\begin{array}{l}
\mathsf{PASSdtv(INTRODUCE)} : \mathsf{np}^?_{\mathsf{by}} \to \mathsf{np}^?_{\mathsf{to}} \to \mathsf{np} \to \mathsf{vp} = \\
\langle\ \lambda b.\lambda t.\lambda s.s \bullet \texttt{was} \bullet \texttt{introduced} \bullet \mathsf{option}(t, \lambda i'.i', \epsilon) \bullet \mathsf{option}(b, \lambda b'.b', \epsilon)\ :\ f^? f^? ff \\
,\ \lambda b.\lambda t.\lambda s.\lambda \mathsf{e}.\ \mathbf{introduce}_{et}(\mathsf{e}) \wedge \mathsf{PAT}(\mathsf{e}, s) \wedge \\
\qquad \mathsf{option}(b\ , \lambda b'.\mathsf{AG}(\mathsf{e}, b') \wedge \mathsf{option}(t\ , \lambda g'.\mathsf{GOAL}(\mathsf{e}, g') \\
\qquad\qquad\qquad\qquad\qquad\qquad , \exists g'.\mathsf{GOAL}(\mathsf{e}, g')) \\
\qquad\qquad , \exists b'.\mathsf{AG}(\mathsf{e}, b') \wedge \mathsf{option}(t\ , \lambda g'.\mathsf{GOAL}(\mathsf{e}, g') \\
\qquad\qquad\qquad\qquad\qquad\qquad , \exists g'.\mathsf{GOAL}(\mathsf{e}, g')))\ :\ e^? e^? eet\ \rangle
\end{array}
$$

Using this passivization operator, sentences with passive verbs can be analyzed without extra passive entries for each combination of missing and present optional arguments. All four possibilities can be derived from a single entry for *introduce*:

1. $\mathsf{EC(PASSdtv(INTRODUCE)(\overline{BY(JOHN)})(\overline{TO(MARY)})(BOB))} : \mathsf{s} =$
   $\langle\ \texttt{Bob} \bullet \texttt{was} \bullet \texttt{introduced} \bullet \texttt{to} \bullet \texttt{Mary} \bullet \texttt{by} \bullet \texttt{John}\ :\ f$
   $,\ \exists \mathsf{e}.\mathbf{introduce'}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, \mathbf{john}_e) \wedge \mathsf{PAT}(\mathsf{e}, \mathbf{bob}_e) \wedge \mathsf{GOAL}(\mathsf{e}, \mathbf{mary}_e)\ :\ t\ \rangle$
2. $\mathsf{EC(PASSdtv(INTRODUCE)(*)(\overline{TO(MARY)})(BOB))} : \mathsf{s} =$
   $\langle\ \texttt{Bob} \bullet \texttt{was} \bullet \texttt{introduced} \bullet \texttt{to} \bullet \texttt{Mary} \bullet \epsilon\ :\ f$
   $,\ \exists \mathsf{e}.\exists b'.\mathbf{introduce'}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, b') \wedge \mathsf{PAT}(\mathsf{e}, \mathbf{bob}_e) \wedge \mathsf{GOAL}(\mathsf{e}, \mathbf{mary}_e)\ :\ t\ \rangle$
3. $\mathsf{EC(PASSdtv(INTRODUCE)(\overline{BY(JOHN)})(*)(BOB))} : \mathsf{s} =$
   $\langle\ \texttt{Bob} \bullet \texttt{was} \bullet \texttt{introduced} \bullet \epsilon \bullet \texttt{by} \bullet \texttt{John}\ :\ f$
   $,\ \exists \mathsf{e}.\mathbf{introduce'}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, \mathbf{john}_e) \wedge \mathsf{PAT}(\mathsf{e}, \mathbf{bob}_e) \wedge \exists g'.\mathsf{GOAL}(\mathsf{e}, g')\ :\ t\ \rangle$
4. $\mathsf{EC(PASSdtv(INTRODUCE)(*)(*)(BOB))} : \mathsf{s} =$
   $\langle\ \texttt{Bob} \bullet \texttt{was} \bullet \texttt{introduced} \bullet \epsilon \bullet \epsilon\ :\ f$
   $,\ \exists \mathsf{e}.\exists b'.\mathbf{introduce'}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, b') \wedge \mathsf{PAT}(\mathsf{e}, \mathbf{bob}_e) \wedge \exists g'.\mathsf{GOAL}(\mathsf{e}, g')\ :\ t\ \rangle$

4.5 Reflexive Optional Arguments

The operator to derive verbs with reflexive optional objects from transitive verbs is defined as:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{REFL} : (\mathsf{np} \to \mathsf{np} \to \mathsf{vp}) \to \mathsf{np}^? \to \mathsf{np} \to \mathsf{vp} =} \\
\langle\ \lambda v.\lambda o.\lambda s.v(\mathsf{option}(o, \lambda o'.o', \epsilon))(s) & :\ (fff)f^? ff \\
,\ \lambda v.\lambda o.\lambda s.\lambda \mathsf{e}.\mathsf{option}(o\ , \lambda o'.v(o'), v(s))(s)(\mathsf{e}) & :\ (eeet)e^? eet\ \rangle
\end{array}
$$

REFL takes a transitive verb and makes its first argument optional in a way that when the argument is missing, the unfilled semantic slot is filled with the entity coming from the subject. Application of REFL to a transitive verb gives a verb with an reflexive optional object.

$$
\begin{array}{l}
\mathsf{REFL(SHAVEtv)} : \mathsf{np}^{?} \to \mathsf{np} \to \mathsf{vp} = \\
\quad \langle\ \lambda o.\lambda s.s \bullet \texttt{shaves} \bullet \mathsf{option}(o, \lambda o'.o', \epsilon)\ \ :\ \ f^{?}ff \\
\quad ,\ \lambda o.\lambda s.\lambda \mathsf{e}.\mathsf{option}(o\ \ ,\lambda o'.\lambda a.\lambda \mathsf{e}.\mathbf{shave}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, a) \wedge \mathsf{PAT}(\mathsf{e}, o') \\
\qquad\qquad\qquad\qquad\quad ,\lambda a.\lambda \mathsf{e}.\mathbf{shave}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, a) \wedge \mathsf{PAT}(\mathsf{e}, s))(s)(\mathsf{e})\ ) \\
\qquad :\ e^{?}eet\ \rangle
\end{array}
$$

Simple examples show that the missing arguments cause the subject to fill the object's unfilled semantic slot :

$$
\begin{array}{l}
\mathsf{EC(REFL(SHAVEtv)(\overline{BOB})(JOHN))} : \mathsf{s} = \\
\quad \langle\ \texttt{John} \bullet \texttt{shaves} \bullet \texttt{Bob} \qquad\qquad :\ f \\
\quad ,\ \exists \mathsf{e}.\mathbf{shave}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, \mathbf{john}_{e}) \wedge \mathsf{PAT}(\mathsf{e}, \mathbf{bob}_{e})\ :\ t\ \rangle \\
\mathsf{EC(REFL(SHAVEtv)(}{*}\mathsf{)(JOHN))} : \mathsf{s} = \\
\quad \langle\ \texttt{John} \bullet \texttt{shaves} \bullet \epsilon \qquad\qquad :\ f \\
\quad ,\ \exists \mathsf{e}.\mathbf{shave}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, \mathbf{john}_{e}) \wedge \mathsf{PAT}(\mathsf{e}, \mathbf{john}_{e})\ :\ t\ \rangle \\
\mathsf{EC(SELF((}\lambda x.\mathsf{SHAVE}(\overline{x})\mathsf{))(JOHN))} : \mathsf{s} = \\
\quad \langle\ \texttt{John} \bullet \texttt{shaves} \bullet \texttt{himself} \qquad\qquad :\ f \\
\quad ,\ \exists \mathsf{e}.\mathbf{shave}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, \mathbf{john}_{e}) \wedge \mathsf{PAT}(\mathsf{e}, \mathbf{john}_{e})\ :\ t\ \rangle
\end{array}
$$

5 General Optionalization

In the previous sections I have demonstrated how option types can be used to treat verbs that allow optional arguments. To restrict the way option types can be used I have suggested the use of optionalization operators. Such an operator can only operate on a single type of verbs. It is of course possible to define such combinations to cover all the cases, but this would require a very large set of operators. A general optionalization procedure is required to introduce optionality in a restricted, but still general way.

A better way to introduce optionality would be to start with a sign without optional arguments, mark the arguments that should be optional and their type of optionality (existential, reflexive, reciprocal, etc), and use a general optionalization procedure to obtain a sign where the marked arguments are optional, and where missing arguments are interpreted the desired way. In this section I will adapt the optionalization procedure given in [Blom et al., 2012] to handle different types of optional arguments.
For example, take the sign for the verb *eat*, defined as an ordinary transitive verb. It takes two obligatory np's, and denotes a ternary relation between two entities and an event.

$$\begin{aligned}&\mathsf{EATtv} : \mathsf{np} \to \mathsf{np} \to \mathsf{vp} = \\ &\langle\ \lambda o.\lambda s.s \bullet \mathsf{eat} \bullet o &&:\ fff \\ &,\ \lambda p.\lambda a.\lambda \mathsf{e}.\mathbf{eat}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, a) \wedge \mathsf{PAT}(\mathsf{e}, p) &&:\ eeet\ \rangle\end{aligned}$$

The verb *eat* actually allows its object to be omitted, in which case it is interpreted as an UO. To obtain an optionalized version of eat, the object is marked for optionalization using the exi marker, which specifies that an unfilled slot should be existentially saturated.

$$\begin{aligned}&\mathsf{EATtv}' : \mathsf{np}^{\mathsf{exi}} \to \mathsf{np} \to \mathsf{vp} = \\ &\langle\ \lambda o.\lambda s.s \bullet \mathsf{eat} \bullet o &&:\ fff \\ &,\ \lambda p.\lambda a.\lambda \mathsf{e}.\mathbf{eat}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, a) \wedge \mathsf{PAT}(\mathsf{e}, p) &&:\ eeet\ \rangle\end{aligned}$$

We need to define how a single UO optional argument is handled in the simplest case. This basic operation is then generalized to a general operation, which can handle more arguments, of which any number can be optional. Application of the general procedure will then results in a sign for *eat* with UO-optionality:

$$\begin{aligned}
&\mathsf{EAT} : \mathsf{np}^? \to \mathsf{np} \to \mathsf{vp} = \\
&\langle\ \lambda o.\lambda s.s \bullet \mathsf{eat} \bullet \mathsf{option}(o, \lambda o'.o', \epsilon)\ \ :\ f^?ff \\
&,\ \lambda p.\lambda a.\lambda \mathsf{e}.\ \ \mathbf{eat}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, a) \wedge \\
&\qquad\qquad\qquad \mathsf{option}(p\ , \lambda p'.\mathsf{PAT}(\mathsf{e}, p'), \exists x.\mathsf{PAT}(\mathsf{e}, x))\ \ :\ e^?eet\ \rangle
\end{aligned}$$

Optionalization is a general procedure to add optionality to non-optional fragments. In the lexicon it is specified which arguments are optional and what type of optionality we are dealing with (existential, reflexive, reciprocal). Saturation operations define how each kind of optional argument deals with missing arguments. An example of a saturation procedure might be to existentially quantify over the missing argument slot of the denotation and fill the missing morpho-syntactic slot with an empty string.

The optionalization procedure given here is only general for existential optional arguments: any argument can be made optional by saturating unfilled slots existentially. For other types of optional arguments I have only given specific cases. Using the operators discussed in section 4.4, specific cases can be treated, such as understood reflexive and reciprocal objects of transitive verbs. Further research is needed to see if and how a general procedure can be given for different types of optional arguments. The procedure is harder to generalize for other kinds of optional arguments, such as understood reflexives and reciprocals, because for these involve other arguments, where existential optional arguments do not.

5.1 Optionalizing a Sign Grammar

The grammar transformation works in two steps: *marking* and *optionalization*. The marking step consists of decorating the arguments that should be optional with the appropriate markers. The assumption is that a marker is part of the lexicon, or is added to a lexicon. The optionalization procedure takes a sign decorated with optionalization markers, and returns a sign where the decorated arguments are optional, where the specific marker determines how missing cases are handled.

5.2 Marking

The first step in optionalizing is to specify which arguments should be optional, and how these should be interpreted when missing. A specification is given by decorating types of signs corresponding to the arguments that should be optional. Such a decoration is called a *marker*, and the application of markers to a lexicon is called a *marking*.

We have already seen that the exi marker is used for existentially filled optional arguments, like the object of *eat*. The exi marker would also be used for the optional prepositional phrases of passives. For verbs that feature a different kind of optionality we use different markers. A causative-unaccusative verb like *sink* has an optional object, that, when missing, causes the deletion of the agent role. To specify this, the refl marker is used:

$$\mathsf{SHAVE} : \mathsf{np}^{\mathsf{refl}} \rightarrow \mathsf{np} \rightarrow \mathsf{s}$$

For each type of optionality there is a different marker, these are $\{\mathsf{exi}, \mathsf{relf}, \mathsf{recip}\}$ which are for existential, reflexive and reciprocal optional arguments respectively.

5.3 Basic Optionalization Procedures

To define how the missing arguments should be handled, *basic* optionalization procedures must be defined. A basic optionalization procedure describes how a missing argument is handled in the simplest case. For each kind of optionality a different basic optionalization procedure is used.

5.3.1 exi: optionality marker for UO's and Passives

exi is the marker used for arguments that should be optional, and that when missing are interpreted as covert existentials.

The corresponding basic optionalization procedure, $\mathsf{opt}_{\mathsf{exi}}$, takes a sign with one argument decorated with a exi optionality marker, and returns a sign where the decorated arguments are optional. $\mathsf{opt}_{\mathsf{exi}}$ is very similar to the UO rule described in section 2. In fact UO has the same effect as marking the object argument of a transitive verb with exi and applying optionalization. Since $\mathsf{opt}_{\mathsf{exi}}$ might be applied multiple times, possibly in combination with other basic optionalization procedures, we need to ensure that the existential closure of the missing argument slot does not get in the way. To do this, the existential closure operator CLOS is used instead of simple existential quantification. The closure operator CLOS existentially saturates the first argument of a function with the narrowest possible scope.

The CLOS operator is defined recursively over the Boolean types.
There are two cases:

(1.) If f is of type $\alpha \to t$, CLOS is simply application of the existential quantifier.
(2.) f is of type $\alpha \to \beta$, where β is Boolean, CLOS is recursively applied so that the first variable is quantified over, while keeping the remaining variables out of the scope of the existential quantifier.

CLOS is defined as:

$$\begin{array}{ll} (1.)\ \mathsf{CLOS}(f : \alpha \to t) & = \exists z_\alpha . f(z) : t \\ \\ (2.)\ \mathsf{CLOS}(f : \alpha \to \beta \to \gamma) & = \lambda y_\beta . \mathsf{CLOS}(\lambda x_\alpha .\, f(x)(y)\,) \\ (\text{where } \gamma \text{ is a Boolean type}) & \end{array}$$

The CLOS operator is best understood by the effect it has: it existentially closes the first argument without obstructing the remaining arguments.

The following example demonstrates the application of CLOS to the 2-place function **like** : *eet*. The first step is to apply the appropriate case of CLOS, as its argument is of type *eet*, the case 2. applies. Following the definition of case 2. of CLOS introduces another instance of CLOS, this time applied to the sub-term $(\lambda x_e . \mathbf{like}(x)(y)\,) : et$. As this term is of type *et*, case 1. of CLOS applies. Application of CLOS introduces the existential quantifier that saturates the first argument. After β-reduction we see that in the resulting term, the first argument to **like** is existentially saturated, and the remaining argument is abstracted over, and bound outside the scope of the quantifier.

$$\begin{array}{rcl} CLOS(\mathbf{like} : eet) & = & \lambda y_e . \mathsf{CLOS}(\lambda x_e . \mathbf{like}(x)(y)\,) \\ & & (\mathsf{CLOS}, \text{ case 1.}) \\ & = & \lambda y_e . \exists z_{\mathsf{e}} . (\lambda x . \mathbf{like}(x)(y))(z) \\ & & (\beta\text{-reduction}) : x \text{ is substituted by } z \\ & = & \lambda y_e . \exists z_{\mathsf{e}} . \mathbf{like}(z)(y) \end{array}$$

The following example demonstrates the application of CLOS to the 3-place function **give** : *eeet*, which procedes similar to the previous example, but with an extra application of $\mathsf{CLOS}$to deal with the extra argument. Once again, in the resulting term, the first argument is existentially saturated, and the remaining arguments are abstracted over and bound outside the scope of the quantifier.

$$\begin{array}{rcl}
\mathsf{CLOS}(\mathbf{give} : eeet) & = & \lambda y_e.\mathsf{CLOS}(\lambda x_e.\mathbf{give}(x)(y)\,) \\
& & (\mathsf{CLOS}, \text{ case 2.}) \\
& = & \lambda y_e.\lambda q_e.\mathsf{CLOS}(\lambda p_{\mathsf{e}}.(\lambda x_e.\mathbf{give}(x)(y)\,)(p)(q)\,) \\
& & (\beta\text{-reduction : } x \text{ is substituted by } p) \\
& = & \lambda y_e.\lambda q_e.\mathsf{CLOS}(\lambda p_{\mathsf{e}}.\mathbf{give}(p)(y)(q)\,) \\
& & (\mathsf{CLOS}, \text{ case 1.}) \\
& = & \lambda y_e.\lambda q_e.\exists z.(\lambda p_\beta.\mathbf{give}(p)(y)(q)\,)(z) \\
& & (\ \beta\text{-reduction : } p \text{ is substituted by } z) \\
& = & \lambda y_e.\lambda q_e.\exists z.\mathbf{give}(z)(y)(q)
\end{array}$$

The existential optionalization procedure $\mathsf{opt} : (\mathsf{a}^{\mathsf{exi}} \to \mathsf{b}) \to (\mathsf{a}^{?} \to \mathsf{b})$ is defined as:

$$\mathsf{opt}\left(\begin{array}{l} \langle\ P : f\gamma \\ ,\ Q : \alpha\beta\ \rangle \end{array} : \mathsf{a}^{\mathsf{exi}} \to \mathsf{b} \right) = \begin{array}{ll} \langle\ \lambda x.\mathsf{option}(x, P,\ P(\epsilon)\,) & : f^{?}\gamma \\ ,\ \lambda x.\mathsf{option}(x, Q, \mathsf{CLOS}(Q)\,) & : \alpha^{?}\beta\ \rangle \end{array} : \mathsf{a}^{?} \to \mathsf{b}$$

(where β is a boolean type.)

The operator opt takes a sign of type $\mathsf{a}^{\mathsf{exi}} \to \mathsf{b}$, of which the morpho-syntactic component is a function from f to γ and the semantic component is a function from α to β, where β is Boolean type, and returns a sign of type $\mathsf{a}^{?} \to \mathsf{b}$. The morpho-syntactic component of the result is a function from an optional string to γ. When the argument is missing, it is resolved with an empty string. The semantic component of the result is a function from an optional α to β. When the argument is missing, it is resolved with the CLOS operator, existentially saturating the first argument.

5.4 Optionalization

5.4.1 The OPT operator

Using a recursive definition, the simple saturation procedure for the different optionality markers can be extended to the general case when possible, by using the appropriate optionalizing operator.

If β is a basic type, and α is not marked for optionalization, f is returned unchanged:

$$\mathsf{OPT}(f : \alpha \to \beta) = f$$

If β is a basic type, and α marked with refl or recip, the corresponding optionalizing operator is applied:

$$\mathsf{OPT}(f : \mathsf{np}^{\mathsf{refl}} \to \mathsf{np} \to \mathsf{vp}) = \mathsf{REFL}(f)$$

$$\mathsf{OPT}(f : \mathsf{np}^{\mathsf{recip}} \rightarrow \mathsf{np} \rightarrow \mathsf{vp}) = \mathsf{RECIP}(f)$$

If β is a basic type, and α is marked for optionalization with exi, the basic optionalization procedure is applied:

$$\mathsf{OPT}(f : \alpha^{\mathsf{exi}} \rightarrow \beta) = \mathsf{opt}(f)$$

If β is a compound type, OPT is recursively applied:

$$\begin{array}{lll} \mathsf{OPT}^*(f : \alpha^{\mathsf{exi}} \rightarrow \beta) & = \mathsf{opt}(& \lambda x.OPT^*(\ f(x)\ )\ ) \\ \mathsf{OPT}^*(f : \alpha \rightarrow \beta) & = & \lambda x.OPT^*(\ f(x)\ ) \end{array}$$

Below is an example of the application of OPT to a marked sign. We start with a non-optional sign for the verb *read*:

$$\begin{array}{ll} \multicolumn{2}{l}{\mathsf{READtv} : \mathsf{np} \rightarrow \mathsf{np} \rightarrow \mathsf{vp} =} \\ \langle\ \lambda o.\lambda s.s \bullet \texttt{read} \bullet o & :\ fff \\ ,\ \lambda p.\lambda a.\lambda \mathsf{e}.\mathbf{read}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, a) \wedge \mathsf{PAT}(\mathsf{e}, p) & :\ eeet\ \rangle \end{array}$$

The first argument should be optional, and when the argument is missing it should be interpreted existentially. We therefore mark the first argument with the exi marker:

$$\begin{array}{ll} \multicolumn{2}{l}{\mathsf{READtv} : \mathsf{np}^{\mathsf{exi}} \rightarrow \mathsf{np} \rightarrow \mathsf{vp} =} \\ \langle\ \lambda o.\lambda s.s \bullet \texttt{read} \bullet o & :\ fff \\ ,\ \lambda p.\lambda a.\lambda \mathsf{e}.\mathbf{read}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, a) \wedge \mathsf{PAT}(\mathsf{e}, p) & :\ eeet\ \rangle \end{array}$$

Next, we apply OPT*, which modifies the sign depending on the optionality markers:

$$\begin{array}{ll} \multicolumn{2}{l}{\mathsf{OPT}(\mathsf{READtv}) : \mathsf{np}^{\mathsf{exi}} \rightarrow \mathsf{np} \rightarrow \mathsf{vp} =} \\ \langle\ \lambda o.\lambda s.s \bullet \texttt{read} \bullet o & :\ fff \\ ,\ \lambda p.\lambda a.\lambda \mathsf{e}.\mathbf{read}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, a) \wedge \mathsf{PAT}(\mathsf{e}, p) & :\ eeet\ \rangle \end{array}$$

As the first argument is marked with exi, the opt operator is applied. After this, no optionality markers remain, so the rest of sign is not affected by OPT$*$

$$\begin{array}{ll} \multicolumn{2}{l}{\mathsf{opt}(\mathsf{READtv}) : \mathsf{np}^{\mathsf{exi}} \rightarrow \mathsf{np} \rightarrow \mathsf{vp} =} \\ \langle\ \lambda o.\lambda s.s \bullet \texttt{read} \bullet o & :\ fff \\ ,\ \lambda p.\lambda a.\lambda \mathsf{e}.\mathbf{read}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, a) \wedge \mathsf{PAT}(\mathsf{e}, p) & :\ eeet\ \rangle \end{array}$$

Application of opt makes the first argument optional. When the argument is missing, it is resolved using an empty string (ϵ) and CLOS.

$$\begin{aligned}
&\mathsf{opt(READtv)} : \mathsf{np}^{\mathsf{exi}} \to \mathsf{np} \to \mathsf{vp} = \\
&\langle\ \lambda x.\mathsf{option}(x, \lambda o.\lambda s.s \bullet \mathtt{read} \bullet o, (\lambda o.\lambda s.s \bullet \mathtt{read} \bullet o)(\epsilon)) \ :\ f^?ff \\
&,\ \lambda x.\mathsf{option}(x\ \ , \lambda p.\lambda a.\lambda \mathsf{e}.\mathbf{read}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, a) \wedge \mathsf{PAT}(\mathsf{e}, p) \\
&\qquad\qquad\qquad , \mathsf{CLOS}(\lambda p.\lambda a.\lambda \mathsf{e}.\mathbf{read}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, a) \wedge \mathsf{PAT}(\mathsf{e}, p))) \ :\ e?eet\ \rangle
\end{aligned}$$

After application of CLOS we obtain the sign for *read* that allows unspecified objects:

$$\begin{aligned}
&\mathsf{opt(READtv)} : \mathsf{np}^{\mathsf{exi}} \to \mathsf{np} \to \mathsf{vp} = \\
&\langle\ \lambda x.\mathsf{option}(x, \lambda o.\lambda s.s \bullet \mathtt{read} \bullet o, \lambda s.s \bullet \mathtt{read} \bullet \epsilon) \ :\ f^?ff \\
&,\ \lambda x.\mathsf{option}(x\ \ , \lambda p.\lambda a.\lambda \mathsf{e}.\mathbf{read}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, a) \wedge \mathsf{PAT}(\mathsf{e}, p) \\
&\qquad\qquad\qquad , \lambda a.\lambda \mathsf{e}.\exists p.\mathbf{read}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, a) \wedge \mathsf{PAT}(\mathsf{e}, p)) \ :\ e?eet\ \rangle
\end{aligned}$$

5.5 The Optionalization Procedure vs Operators

Suppose PASStv' is a rule that maps transitive verbs to their non-optional passive form. It rearranges the word order, adds the auxiliary verb `was`, and changes an np argument into a $\mathsf{np}_{\mathsf{by}}$ argument, but does not yet optionalize any arguments.

$$\begin{aligned}
&\mathsf{PASStv}' : (\mathsf{np} \to \mathsf{np} \to \mathsf{vp}) \to \mathsf{np}_{\mathsf{by}} \to \mathsf{np} \to \mathsf{vp} = \\
&\langle\ \lambda v.\lambda o.\lambda s.v(o)(s \bullet \mathtt{was}) \ :\ (fff)fff \\
&,\ \lambda v.\lambda o.\lambda s.\lambda \mathsf{e}.v(s)(o)(\mathsf{e}) \ :\ (eeet)eeet\ \rangle
\end{aligned}$$

So far this sign is fairly simple, but the *by* phrase argument is not optional yet. Optionality can be added to this rule using the exi marker:

$$\begin{aligned}
&\mathsf{PASStv'exi} : (\mathsf{np} \to \mathsf{np} \to \mathsf{vp}) \to \mathsf{np}_{\mathsf{by}}^{\mathsf{exi}} \to \mathsf{np} \to \mathsf{vp} = \\
&\langle\ \lambda v.\lambda o.\lambda s.v(o)(s \bullet \mathtt{was}) \ :\ (fff)fff \\
&,\ \lambda v.\lambda o.\lambda s.\lambda \mathsf{e}.v(s)(o)(\mathsf{e}) \ :\ (eeet)eeet\ \rangle
\end{aligned}$$

Application of OPT results in a sign equivalent to the PASStv operator defined in the previous chapter:

$$\begin{aligned}
&\mathsf{PASStv} : (\mathsf{np} \to \mathsf{np} \to \mathsf{vp}) \to \mathsf{np}_{\mathsf{by}}^{?} \to \mathsf{np} \to \mathsf{vp} = \\
&\langle\ \lambda v.\lambda o.\lambda s.v(\mathsf{option}(o, \lambda o'.o', \epsilon))(s \bullet \mathtt{was}) \ :\ (fff)f^?ff \\
&,\ \lambda v.\lambda o.\lambda s.\lambda \mathsf{e}.\mathsf{option}(o\ , \lambda o'.v(s)(o')(\mathsf{e}), \exists o'.v(s)(o')(\mathsf{e})) \ :\ (eeet)e^?eet\ \rangle
\end{aligned}$$

The advantage of the optionalization procedure is that multiple optional arguments are much easier to define. To define a passivization operator for a ditransitive verb, which has two optional arguments, all we need to do is add two exi markers to the abstract type:

$$\begin{array}{l}\mathsf{PASSdtv'exi} : (\mathsf{np} \to \mathsf{np} \to \mathsf{np} \to \mathsf{vp}) \to \mathsf{np}_{\mathsf{by}}^{\mathsf{exi}} \to \mathsf{np}_{\mathsf{to}}^{\mathsf{exi}} \to \mathsf{np} \to \mathsf{vp} = \\ \langle\ \lambda v.\lambda b.\lambda t.\lambda s.v(b)(t)(s \bullet \mathtt{was}) :\ (ffff)ffff \\ ,\ \lambda v.\lambda b.\lambda t.\lambda s.\lambda \mathsf{e}.v(s)(t)(b)(\mathsf{e}) :\ (eeeet)eeeet\ \rangle\end{array}$$

Application of OPT to this marked sign gives us the sign for passivization of ditransitive verb:

$$\begin{array}{l}\mathsf{PASSdtv} : (\mathsf{np} \to \mathsf{np}^{?} \to \mathsf{np} \to \mathsf{vp}) \to \mathsf{np}_{\mathsf{by}}^{?} \to \mathsf{np}_{\mathsf{to}}^{?} \to \mathsf{np} \to \mathsf{vp} = \\ \langle\ \lambda v.\lambda b.\lambda t.\lambda s.v(\mathsf{option}(b, \lambda b'.b', \epsilon))(t)(s \bullet \mathtt{was}) :\ (ff^{?}ff)f^{?}f^{?}ff \\ ,\ \lambda v.\lambda b.\lambda t.\lambda s.\lambda \mathsf{e}.\mathsf{option}(b\ ,\lambda b'.v(s)(t)(b')(\mathsf{e}) \quad :\ (ee^{?}eet)e^{?}e^{?}eet\ \rangle \\ \qquad\qquad\qquad\qquad ,\exists b'.v(s)(t)(b')(\mathsf{e}))\end{array}$$

This complex sign is equivalent to the passivization operator given in section 4.4.2, but is derived from a much simpler sign and a few markers. The optionalization procedure introduces the nested option operators that are needed to deal with the optional arguments.

The optionalization procedure provides a general way to add existential optionality to any sign. Extending this operation to a sign grammar is trivial. A sign grammer is just a list of signs and a target type, so optionalizing a sign grammar is done by marking and optionalizing each of the signs in that list. The optionalization of grammars gives us a modular way to deal with optionality, deriving complex grammars with optionality from simple grammars without optionality, simply by placing markers on the types.

5.6 Optionalization and Optional Verb Classes

Using these markers, we can refine table 3. The previous version showed us the number and type of arguments required by each verb class, and which of these arguments are optional. Using the optionality markers, we can also specify for each verb class how these optional arguments are interpreted when omitted.

Verb class	Valence	Abstract type	Examples
intransitive	1	$\mathsf{np} \to \mathsf{vp}$	*run,fall*
transitive	2	$\mathsf{np} \to \mathsf{np} \to \mathsf{vp}$	*build, hit*
transitive with unspecified object	1 or 2	$\mathsf{np}^{\mathsf{exi}} \to \mathsf{np} \to \mathsf{vp}$	*eat, read*
transitive with reflexive o. object	1 or 2	$\mathsf{np}^{\mathsf{refl}} \to \mathsf{np} \to \mathsf{vp}$	*shave, dress*
transitive with reciprocal o. object	1 or 2	$\mathsf{np}^{\mathsf{recip}} \to \mathsf{np} \to \mathsf{vp}$	*kiss, meet*
passive transitive	1 or 2	$\mathsf{np}^{\mathsf{exi}}_{\mathsf{by}} \to \mathsf{np} \to \mathsf{vp}$	*was build*
ditransitive with opt. indirect obj.	2 or 3	$\mathsf{np} \to \mathsf{np}^{\mathsf{exi}} \to \mathsf{np} \to \mathsf{vp}$	*give*
ditransitive with opt *to* phrase	2 or 3	$\mathsf{np} \to \mathsf{np}^{\mathsf{exi}}_{\mathsf{to}} \to \mathsf{np} \to \mathsf{vp}$	*introduce*
passive ditransitive	1,2 or 3	$\mathsf{np}^{\mathsf{exi}}_{\mathsf{by}} \to \mathsf{np}^{\mathsf{exi}}_{\mathsf{to}} \to \mathsf{np} \to \mathsf{vp}$	*was given*

Table 4: Types for verb classes with optionality markers.

6 Mixing Optional and Non-optional Signs

In the last two chapters I have explained how option types can be used to treat verbs with optional arguments and how to introduce optionality with a general procedure. This optionalization procedure provides a structured method to obtain a grammar with optional arguments from a grammar with only obligatory arguments. Many constructions in language do not feature optional arguments, and many of these construction may need to interact with verbs with optional arguments. In this section I will show that such interactions are possible, and that their possibility follows from the rules of ACG with option types.

6.1 Coordination

Suppose we want to write a lexicon that allows us to derive the following sentences.

1. John works and sleeps.
2. John built and sold the car.
3. John worked and read.
4. John read and read a book.

In sentence (3.) and (4.) intransitive and transitive verbs are coordinated with ambitransitive verbs. Assume that we have the following types for the signs needed to analyze the sentences:

$\mathsf{JOHN}, \mathsf{CAR}$	np
$\mathsf{WORK}, \mathsf{SLEEP}$	$\mathsf{np} \to \mathsf{vp}$
$\mathsf{SELL}, \mathsf{BUILD}$	$\mathsf{np} \to \mathsf{np} \to \mathsf{vp}$

In general, *and* coordinates expressions with the same category. In categorial grammar, coordinating conjunctions have type $X \to X \to X$, where X can be any suitable category. For intransitive ($\mathsf{np} \to \mathsf{s}$) and transitive verbs ($\mathsf{np} \to \mathsf{np} \to \mathsf{s}$), the following lexical entries would be sufficient to deal with coordination, and derive the sentences.

$\mathsf{AND}_{\mathsf{iv}}$	$(\mathsf{np} \to \mathsf{vp}) \to (\mathsf{np} \to \mathsf{vp}) \to (\mathsf{np} \to \mathsf{vp})$
$\mathsf{AND}_{\mathsf{tv}}$	$(\mathsf{np} \to \mathsf{np} \to \mathsf{vp}) \to (\mathsf{np} \to \mathsf{np} \to \mathsf{vp}) \to (\mathsf{np} \to \mathsf{np} \to \mathsf{vp})$

Let iv be an abbreviation for $\mathsf{np} \to \mathsf{vp}$, tv for $\mathsf{np} \to \mathsf{np} \to \mathsf{vp}$ and av for $\mathsf{np}^? \to \mathsf{np} \to \mathsf{vp}$. Using the following types for *and*, we can analyze sentence d1. and d2.:

d1 $(\mathsf{AND}_{\mathsf{iv}\to\mathsf{iv}\to\mathsf{iv}}(\mathsf{WORK}_{\mathsf{iv}})(\mathsf{SLEEP}_{\mathsf{iv}}))(\mathsf{JOHN})$

d2 $(\mathsf{AND}_{\mathsf{tv}\to\mathsf{tv}\to\mathsf{tv}}(\mathsf{BUY}_{\mathsf{tv}})(\mathsf{SELL}_{\mathsf{tv}}))(\mathsf{CAR})(\mathsf{JOHN})$

But what about coordinating a verb of type $\mathsf{np} \to \mathsf{np} \to \mathsf{vp}$ and verb of type $\mathsf{np}^? \to \mathsf{np} \to \mathsf{vp}$, such as READ and WORK? Because coordinating conjunctions require that their arguments have the same type, transitive verbs can be coordinated with transitive verbs, intransitives with intransitives, but transitive or intransitive verbs cannot directly be coordinated with ambitransitive verbs. Combining them would lead to a type mismatch:

d3 $\underbrace{(\mathsf{AND}_{\mathsf{iv}\to\mathsf{iv}\to\mathsf{iv}}(\mathsf{WORK}_{iv})(\mathsf{WROTE}_{av}))}_{\text{type mismatch } \mathsf{iv}\neq\mathsf{av}}(\mathsf{JOHN})$

d4 $\underbrace{(\mathsf{AND}_{\mathsf{tv}\to\mathsf{tv}\to\mathsf{tv}}(\mathsf{BUY}_{tv})(\mathsf{WROTE}_{av}))}_{\text{type mismatch } \mathsf{tv}\neq\mathsf{av}}(\mathsf{JOHN})$

Another example of such a mismatch occurs with reflexivisation. In Montagovian treatments and categorial grammars, reflexivisation is modeled as a function from transitive verbs to intransitive verbs, where the agent and patient refer to the same entity.[14]

[14]see section 4.3.3 for an explanation of SELF

$$\begin{aligned}&\mathsf{SELF} : (\mathsf{np} \to \mathsf{np} \to \mathsf{vp}) \to \mathsf{np} \to \mathsf{vp} = \\ &\langle\ \lambda v.\lambda s.v(\texttt{himself})(s) \ :\ (fff)ff \\ &,\ \lambda v.\lambda a.\lambda \mathsf{e}.v(a)(a)(\mathsf{e}) \ \ :\ (eeet)eet\ \rangle\end{aligned}$$

Using the definition above it is not possible to use a reflexive object in combination with a transitive verb, like *sink*, with optional arguments, because their types do not match. SELF expects an argument of type $\mathsf{np} \to \mathsf{np} \to \mathsf{vp}$, but SINK is of type $\mathsf{np}^? \to \mathsf{np} \to \mathsf{vp}$.

- The car sold itself.
  $(\mathsf{SELF}_{\mathsf{tv}\to\mathsf{iv}}(\mathsf{SELL}_{\mathsf{tv}}))(\mathsf{CAR})$
- The ship sunk itself.

  $\underbrace{(\mathsf{SELF}_{\mathsf{tv}\to\mathsf{iv}}(\mathsf{SINK}_{\mathsf{av}}))}_{\text{type mismatch } tv\neq av}(\mathsf{SHIP})$

Many other constructions, like adverbs, reciprocals, passivization, and many more, require similar interactions between signs with and without optional arguments. It is of course possible to add extra entries to cover all possible combinations of optional and non-optional arguments, but again the number of combinations is vast. A general way of letting signs with optional arguments interact with non-optional components of the grammar is required, so that these components can interact with the signs with optional arguments.
Luckily, in ACG with option types, any optional argument can be omitted or made obligatory. A transitive with an optional argument like $\mathsf{READ} : \mathsf{np}^? \to \mathsf{np} \to \mathsf{vp}$ can behave as an intransitive $\mathsf{np} \to \mathsf{vp}$ or a transitive verb $\mathsf{np} \to \mathsf{np} \to \mathsf{vp}$ where needed. This casting of an optional argument to its specific non-optional variants is a feature of the ACG with option types system.

6.2 Getting rid of optional arguments

Changing an optional argument of some verb into an obligatory argument is conceptually very simple: we need to remove the question mark from the type. This can always be done by hypothetical reasoning: assume some value, apply option injection to the assumed value, use the injected value as the argument of the verb, and then discharge the hypothesis by abstracting over the assumed value.

If we have a sign $\mathsf{A} : p^? \to q$, making the optional argument obligatory results in a sign $\mathsf{A}' : p \to q$
First, we assume a value $(x : p)$, make it optional using option injection $(\overline{x} : p^?)$, apply it to $(\mathsf{A}(\overline{x}) : q)$, and finally discharge the assumed

value by binding it using a lambda ($\lambda x.\mathsf{A}(\overline{x}) : p \to q$)).

This process is part of the grammatical system: application, option injection and hypothetical reasoning are all part of the ACG system. In some cases it is clearer to abbreviate the steps as a sign. Below is an example of such a sign, that when applied to a transitive verb with an optional object, returns a transitive verb.

$$\begin{aligned}\mathsf{RequireObj}_{\mathsf{tv}} = \ & \mathsf{RequireObj} : (\mathsf{np}^? \to \mathsf{np} \to \mathsf{vp}) \to \mathsf{np} \to \mathsf{np} \to \mathsf{vp} = \\ & \langle\ \lambda v.\lambda x.\lambda y.v(\overline{x})(y) :\ (f^? ff)fff \\ & ,\ \lambda v.\lambda x.\lambda y.v(\overline{x})(y) :\ (e^? eet)eeet\ \rangle\end{aligned}$$

RequireObj takes an ambitransitive verb sign with an optional first argument, and returns a sign where that argument is no longer optional. Below is a demonstration of RequireObj applied to the sign of the ambitransitive verb eat. Note that the result of this has the form of an ordinary transitive verb.

$$\begin{aligned}& \mathsf{READ} : \mathsf{np}^? \to \mathsf{np} \to \mathsf{vp} = \\ & \langle\ \lambda o.\lambda s.s \bullet \mathtt{read} \bullet \mathsf{option}(o, \lambda o'.o', \epsilon)\ :\ f^? ff \\ & ,\ \lambda p.\lambda a.\lambda \mathsf{e}.\ \mathbf{read}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, a) \wedge \\ & \qquad\qquad \mathsf{option}(p\ , \lambda p'.\mathsf{PAT}(\mathsf{e}, p') \\ & \qquad\qquad\qquad , \exists p'.\mathsf{PAT}(\mathsf{e}, p'))\ :\ e^? eet\ \rangle \\ & \mathsf{RequireObj}(\mathsf{READ}) : \mathsf{np} \to \mathsf{np} \to \mathsf{vp} = \\ & \langle\ \lambda x.\lambda y.y \bullet \mathtt{read} \bullet x \qquad\qquad\qquad :\ fff \\ & ,\ \lambda x.\lambda y.\lambda \mathsf{e}.\mathbf{read}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, y) \wedge \mathsf{PAT}(\mathsf{e}, x)\ :\ eeet\ \rangle\end{aligned}$$

Dropping an optional argument is even simpler. Here we only have to fill the optional slot with $*$. If the optional argument is not the first argument, we can introduce variables to saturate the other arguments, then fill the optional argument with $*$, and then bind these variables with lambda's. This is again part of ACG's general Hypothetical Reasoning.

If we have a sign $\mathsf{A} : p^? \to q$, dropping the argument results in a $\mathsf{A}^* : q$ All that is needed is to fill the optional argument with the universal filler $\mathsf{A}(*) = q$.
Similar to the RequireObj combinator that makes an optional object required, there is a combinator that drops optional objects:

$$\begin{aligned}& \mathsf{DropObj} : (\mathsf{np}^? \to \mathsf{np} \to \mathsf{vp}) \to \mathsf{np} \to \mathsf{vp} = \\ & \langle\ \lambda v.\lambda y.v(*)(y) :\ (f^? ff)ff \\ & ,\ \lambda v.\lambda y.v(*)(y) :\ (e^? eet)eet\ \rangle\end{aligned}$$

Application of DropObj to READ gives the intransitive form of *read*:

$$\begin{array}{ll} \mathsf{DropObj}(\mathsf{READ}) : \mathsf{np} \rightarrow \mathsf{vp} = & \\ \langle\ \lambda y.y \bullet \mathtt{read} \bullet \epsilon & :\ ff \\ ,\ \lambda y.\lambda \mathsf{e}.\mathbf{read}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, y) \wedge \exists p'.\mathsf{PAT}(\mathsf{e}, p') & :\ eet\ \rangle \end{array}$$

Since such argument-requiring and argument-dropping combinators are only composed of lambda abstraction, function application and option injection such signs are combinators. A combinator is a lambda term in which all variables are bound and that does not contain any constants, and is therefore derivable. Because of this, there is no need to define these combinators in the lexicon. These combinators are simply abbreviations for common patterns used when working signs, which are derivable from the rules of ACG with option types.

Any optional argument can be made obligatory or can be dropped at any time, using a combination of hypothetical reasoning and option injection, or by application of the universal filler $*$. The examples above feature the simplest case, when there is only a single argument. To extend the procedure to multiple arguments, we simply use hypothetical reasoning to temporarily saturate the other arguments, and discharge the hypothesis when the optional argument has been dealt with.

6.3 Transforming Optional Grammars to Non-Optional Grammars

Any sign grammar that has optional arguments can be transformed into a grammar without optional arguments that generates the same language. This transformation can easily be defined by dropping and requiring each optional argument of each sign, in effect enumerating all the combinations of provided and missing arguments. This deoptionalization procedure shows that we can eliminate optional arguments in a grammar, at the cost of more lexical entries. There are three reasons for giving this de-optionalization procedure: The first reason is that sometimes we need to eliminate the optional arguments in a sign so can it can serve as an argument to a sign that expects a non-optional sign as its arugment. We have seen some examples of this in section 6. While this is already possible in ACG with option types, using Hypothetical Reasoning an option injection, the deoptionalization procedure abbreviates the intermediate steps, which simplifies dealing with optional arguments. The second reason is that it allows us to compare signs with optional arguments or grammars with optionality to other formalisms that do not accomodate optionality. The third reason is to show the efficiency that can be achieved using option types: a de-optionalized grammar will generate the same language as a grammar with optional entries, but requires much more lexical entries.

6.4 Basic Deoptionalization Procedure

To enumerate all possibilities, the implicit process of making optional argument required *(using Hyp. Reasoning an option injection)* or getting rid *(using the null option as argument)* of them, must be made explicit. This is done using markers and basic procedures, much like the optionalization procedure. Note that the deoptionalization procedure is of a different nature than the optionalization procedure, as the enumeration is already part of the grammatical system, as we have seen in the previous section.

There are 3 things we can do with an optional argument: we can leave it as it is, it can be made obligatory, or it can be dropped. The first option is trivial, for the other two there are two markers: $\downarrow$ which marks an optional argument for removal, and $\uparrow$ which marks that an optional argument should be made obligatory. These two de-optionalization markers, $D = \{\downarrow, \uparrow\}$, can only mark optional arguments. Similar to how for each optionalization marker M there is a basic optionalization procedure opt^M, there is a basic de-optionalization procedure deopt_D for each de-optionalization marker D. The basic de-optionalization procedures are very simple:

$\mathsf{deopt}_\downarrow \;= \lambda f.f(*) : (a^? \to b) \to b$
fill the slot with $*$
$\mathsf{deopt}_\uparrow \;= \lambda f.\lambda x.f(\overline{x}) : (a^? \to b) \to a \to b$
make the argument obligatory,
using option injection and hyp. reasoning

The general de-optionalization procedure is defined as:

$\mathsf{DEOPT}(f : \alpha) \;= f$
(if α is a basic type or contains no D markers)
$\mathsf{DEOPT}(f : \alpha \to b) \;= \lambda x.\mathsf{DEOPT}(f(x))$
$\mathsf{DEOPT}(f : \alpha^? \to b) \;= \lambda x.\mathsf{DEOPT}(f(x))$
$\mathsf{DEOPT}(f : \alpha^{?D} \to b) \;= \mathsf{deopt}_D(\,\lambda x.\mathsf{DEOPT}(f(x))\,)$

The effect of DEOPT to a marked function is that all marked optional arguments will be either removed or made obligatory, depending on the markings. If f is of a basic type, or contains no marked optional arguments (no D markers), then DEOPT will return f unchanged. If

f's first argument is not marked, that argument will not be modified, and DEOPT will recursively apply to the next argument. If f's first argument is marked with a de-optionalization marker D, that argument will not be modified using deopt_D, and DEOPT will recursively apply to the next argument.

In the following example, DEOPT is applied to the sign for passive *introduce*, which is marked such that the optional *by* phrase argument will be removed, and the *to* phrase argument will be made required.

$$\begin{aligned}
& \mathsf{DEOPT}(\,\mathsf{INTROpass} : \mathsf{np}_{\mathsf{by}}^{?\downarrow} \to \mathsf{np}_{\mathsf{to}}^{?\uparrow} \to \mathsf{np} \to \mathsf{vp}\,) \\
= \quad & \mathsf{deopt}_{\downarrow}(\,\lambda x.\mathsf{DEOPT}(\,\mathsf{INTROpass}(x) : \mathsf{np}_{\mathsf{to}}^{\uparrow?} \to \mathsf{np} \to \mathsf{vp}\,)\,)\,) \\
= \quad & \mathsf{deopt}_{\downarrow}(\,\lambda x.\mathsf{deopt}_{\uparrow}(\,\lambda y.\mathsf{INTROpass}(x)(y) : \mathsf{np} \to \mathsf{vp}\,)\,) \\
= \quad & \mathsf{deopt}_{\downarrow}(\,\lambda x.\lambda y.\mathsf{INTROpass}(x)(\overline{y}) : \mathsf{np}_{\mathsf{to}} \to \mathsf{np} \to \mathsf{vp}\,) \\
= \quad & \lambda y.\mathsf{INTROpass}(*)(\overline{y}) : \mathsf{np}_{\mathsf{to}} \to \mathsf{np} \to \mathsf{vp}
\end{aligned}$$

Using DEOPT, $\uparrow$ and $\downarrow$ can be used as abbreviations for dropping and requiring optional arguments. In fact, using de-optionalization markers, combinators like $\mathsf{DropObj}$ can be derived from a trivial sign with the appropriate markers. Suppose we have a sign $\mathsf{DropObj}'$, of which all components denote the identity function. Now we add the $\downarrow$ marker to the optional argument:

$$\mathsf{DropObj}' : (\mathsf{np}? \to \mathsf{np} \to \mathsf{vp}) \to \mathsf{np}^{\downarrow?} \to \mathsf{np} \to \mathsf{vp} = \langle id, id \rangle$$

Application of DEOPT to this sign results in the optional object dropping combinator $\mathsf{DropObj}$

$$\mathsf{DEOPT}(\mathsf{DropObj}') = \mathsf{DropObj}$$

This demonstrates that adding de-optionalization markers and applying DEOPT can achieve the same as explicitly removing optional arguments, but in a much more concise way.

6.5 Enumeration as a Grammar Transformation

Using DEOPT we can define a grammar transformation that gets rid of all the optional arguments in the source grammar, while preserving the language it generates.

Let $allMarkings(f : t)$ function that takes a sign f of type t and returns the set of signs with all possible markings of the optional arguments of f.

There is a simple relation between the number of optional arguments and the number of de-optionalization markings: if f has n optional arguments, $|allMarkings(f)|$ will be 2^n

Example:

$$
\begin{array}{ll}
\multicolumn{2}{l}{allMarkings(\mathsf{INTROpass} : \mathsf{np}^{?}_{\mathsf{by}} \to \mathsf{np}^{?}_{\mathsf{to}} \to \mathsf{np} \to \mathsf{vp}\,)} \\
= \{ & \mathsf{INTROpass} : \mathsf{np}^{\downarrow ?}_{\mathsf{by}} \to \mathsf{np}^{\downarrow ?}_{\mathsf{to}} \to \mathsf{np} \to \mathsf{vp} \\
, & \mathsf{INTROpass} : \mathsf{np}^{\downarrow ?}_{\mathsf{by}} \to \mathsf{np}^{\uparrow ?}_{\mathsf{to}} \to \mathsf{np} \to \mathsf{vp} \\
, & \mathsf{INTROpass} : \mathsf{np}^{\uparrow ?}_{\mathsf{by}} \to \mathsf{np}^{\downarrow ?}_{\mathsf{to}} \to \mathsf{np} \to \mathsf{vp} \\
, & \mathsf{INTROpass} : \mathsf{np}^{\uparrow ?}_{\mathsf{by}} \to \mathsf{np}^{\uparrow ?}_{\mathsf{to}} \to \mathsf{np} \to \mathsf{vp} \\
\} &
\end{array}
$$

By applying DEOPT to all the marked versions of a sign s, all possible de-optionalized versions of s are obtained. This procedure is called non-optionals, and is defined as:

$$\mathsf{non\text{-}optionals}(s) = \{DEOPT(s') \mid s' \in allMarkings(s)\}$$

Let $S = \langle I, S, d\rangle$ be a sign grammar, and let S be the signs of G. The de-optionalized signs of S, $D(S)$, are given by the following procedure:

$$D(\langle I, S, d\rangle) = \langle I, \{s' \mid s \in S, s' \in \mathsf{non\text{-}optionals}(s)\}, D\rangle$$

For each sign s of the signs S of grammar G, enumerate all possible de-optionalization markings s'. Apply DEOPT to these marked signs, and collect all the results. The resulting grammar no longer contains optional arguments, but may be much larger. For each sign with n optional arguments in the source grammar, there will be 2^n signs in the de-optionalized grammar.

6.6 From Non-Optional Grammars to Optional Grammars and back

To demonstrate the optionalization an enumeration procedure, I will show a simple grammar without optionality can be extended with optional arguments, and how to transform this extended grammar back to a grammar without optional arguments:

The first step is *marking*: certain arguments that should be optional are annotated using the optionalization markers which determine in

Table 5: Non-optional lexicon

Signs	Abstract type	Concrete components
$\mathsf{shave_{tv}}$	$\mathsf{np} \to \mathsf{np} \to \mathsf{vp}$	$\langle\ \lambda o.\lambda s.s \bullet \texttt{shaves} \bullet o$
		$,\ \lambda p.\lambda a.\lambda e.\mathbf{shave}_{et}(e) \wedge \mathsf{AG}(e,a) \wedge \mathsf{PAT}(e,p)\ \rangle$
$\mathsf{eat_{tv}}$	$\mathsf{np} \to \mathsf{np} \to \mathsf{vp}$	$\langle\ \lambda o.\lambda s.s \bullet \texttt{eat} \bullet o$
		$,\ \lambda p.\lambda a.\lambda e.\mathbf{eat}_{et}(e) \wedge \mathsf{AG}(e,a) \wedge \mathsf{PAT}(e,p)\ \rangle$
$\mathsf{intro_{pass}}$	$\mathsf{np_{by}} \to \mathsf{np_{to}} \to \mathsf{np} \to \mathsf{vp}$	$\langle\ \lambda b.\lambda t.\lambda s.s \bullet \texttt{was-introduced} \bullet b \bullet t$
		$,\ \lambda a.\lambda g.\lambda p.\lambda e.\ \mathbf{introduce}_{et}(e) \wedge \mathsf{PAT}(e,p)$
		$\wedge \mathsf{AG}(e,a) \wedge \mathsf{GOAL}(e,g)\ \rangle$

what way the argument will be optional. A marked lexicon contains information about which arguments are optional and how they should be interpreted when missing.

Table 6: Marked lexicon:

Signs	Abstract type	Concrete components
$\mathsf{shave_{tv}}$	$\mathsf{np^{refl}} \to \mathsf{np} \to \mathsf{vp}$	$\langle\ \lambda o.\lambda s.s \bullet \texttt{shaves} \bullet o$
		$,\ \lambda p.\lambda a.\lambda e.\mathbf{shave}_{et}(e) \wedge \mathsf{AG}(e,a) \wedge \mathsf{PAT}(e,p)\ \rangle$
$\mathsf{eat_{tv}}$	$\mathsf{np^{exi}} \to \mathsf{np} \to \mathsf{vp}$	$\langle\ \lambda o.\lambda s.s \bullet \texttt{eat} \bullet o$
		$,\ \lambda p.\lambda a.\lambda e.\mathbf{eat}_{et}(e) \wedge \mathsf{AG}(e,a) \wedge \mathsf{PAT}(e,p)\ \rangle$
$\mathsf{intro_{pass}}$	$\mathsf{np^{exi}_{by}} \to \mathsf{np^{exi}_{to}} \to \mathsf{np} \to \mathsf{vp}$	$\langle\ \lambda b.\lambda t.\lambda s.s \bullet \texttt{was-introduced} \bullet b \bullet t$
		$,\ \lambda a.\lambda g.\lambda p.\lambda e.\ \mathbf{introduce}_{et}(e) \wedge \mathsf{PAT}(e,p) \wedge$
		$\mathsf{AG}(e,a) \wedge \mathsf{GOAL}(e,g)\ \rangle$

By using the optionalization procedure, the information about the optionality of the arguments is taken from the level of types to the level of terms, by making argument optional and inserting option operators to deal with these optional arguments.

The optionalized entries can now deal with optional arguments, and missing arguments will be resolved according to the basic optionalization procedure used to make the argument optional. ACG with option types can account for interactions between signs with and without optional arguments: argument dropping and argument requiring are baked in the grammatical system, but for comparison with non-optional frameworks, the enumeration transformations can be applied to give an equivalent non-optional grammar.

The enumeration transformation is performed by decorating the optional entries with all possible deoptionalization markers, and then

Table 7: The lexicon after optionalization:

Signs	Abstract type	Concrete components
shave	$\mathsf{np}^? \to \mathsf{np} \to \mathsf{vp}$	$\langle\ \lambda o.\lambda s.s \bullet \texttt{shaves} \bullet \mathsf{option}(o, \lambda o'.o', \epsilon)$ $,\ \ \lambda p.\lambda a.\lambda e.\mathbf{shave}_{et}(e) \wedge \mathsf{AG}(e, a)$ $\wedge\mathsf{option}(p, \lambda p'.\mathsf{PAT}(e, p'), \mathsf{PAT}(e, a))\rangle$
eat	$\mathsf{np}^? \to \mathsf{np} \to \mathsf{vp}$	$\langle\ \lambda o.\lambda s.s \bullet \texttt{eat} \bullet \mathsf{option}(o, \lambda o'.o', \epsilon)$ $,\ \ \lambda p.\lambda a.\lambda e.\mathbf{eat}_{et}(e) \wedge \mathsf{AG}(e, a)$ $\wedge\mathsf{option}(p, \lambda p'.\mathsf{PAT}(e, p'), \exists p'.\mathsf{PAT}(e, p'))\rangle$
$\mathsf{intro}_{\mathsf{pass}}$	$\mathsf{np}^?_{\mathsf{by}} \to \mathsf{np}^?_{\mathsf{to}} \to \mathsf{np} \to \mathsf{vp}$	$\langle\ \lambda b.\lambda t.\lambda s.s\ \bullet\texttt{was-introduced}$ $\bullet\mathsf{option}(b, \lambda b'.b', \epsilon)$ $\bullet\mathsf{option}(t, \lambda t'.t', \epsilon)$ $,\ \ \lambda a.\lambda g.\lambda p.\lambda e.\mathbf{introduce}_{et}(e) \wedge \mathsf{PAT}(e, p)$ $\wedge\mathsf{option}(a, \lambda a'.\mathsf{AG}(e, a'), \exists a'.\mathsf{AG}(e, a'))$ $\wedge\mathsf{option}(g, \lambda g'.\mathsf{GOAL}(e, g'), \exists g'.\mathsf{GOAL}(e, g'))\rangle$

applying the DEOPT procedure to each marked sign. The effect of the transformation is that each possible combination of providing an optional argument with an argument is enumerated explicitly in the lexicon. The enumeration transformation demonstrates that a lexicon with optional arguments can be 'compiled' to a lexicon without optional arguments, eliminating the need for option types, at the cost of multiple lexical entries.

Table 8: Deoptionalized lexicon

Signs	Abstract type	Concrete components
$\mathsf{shave_{iv}}$	$\mathsf{np} \to \mathsf{vp}$	$\langle\ \lambda o.\lambda s.s \bullet \texttt{shaves}$ $,\ \lambda a.\lambda e.\mathbf{shave}_{et}(e) \wedge \mathsf{AG}(e,a) \wedge \mathsf{PAT}(e,a)\ \rangle$
$\mathsf{shave_{tv}}$	$\mathsf{np} \to \mathsf{np} \to \mathsf{vp}$	$\langle\ \lambda o.\lambda s.s \bullet \texttt{shaves} \bullet o$ $,\ \lambda p.\lambda a.\lambda e.\mathbf{shave}_{et}(e) \wedge \mathsf{AG}(e,a) \wedge \mathsf{PAT}(e,p)\ \rangle$
$\mathsf{eat_{iv}}$	$\mathsf{np} \to \mathsf{vp}$	$\langle\ \lambda s.s \bullet \texttt{eat}$ $,\ \lambda a.\lambda e.\mathbf{eat}_{et}(e) \wedge \mathsf{AG}(e,a) \wedge \exists p.\mathsf{PAT}(e,p)\ \rangle$
$\mathsf{eat_{tv}}$	$\mathsf{np} \to \mathsf{np} \to \mathsf{vp}$	$\langle\ \lambda o.\lambda s.s \bullet \texttt{eat} \bullet o$ $,\ \lambda p.\lambda a.\lambda e.\mathbf{eat}_{et}(e) \wedge \mathsf{AG}(e,a) \wedge \mathsf{PAT}(e,p)\ \rangle$
intro	$\mathsf{np} \to \mathsf{vp}$	$\langle\ \lambda b.\lambda s.s \bullet \texttt{was-introduced}$ $,\ \lambda p.\lambda e.\ \mathbf{introduce}_{et}(e) \wedge \mathsf{PAT}(e,p)$ $\wedge \exists a.\mathsf{AG}(e,a)$ $\wedge \exists g.\mathsf{GOAL}(e,g)\ \rangle$
intro_{by}	$\mathsf{np_{by}} \to \mathsf{np} \to \mathsf{vp}$	$\langle\ \lambda b.\lambda s.s \bullet \texttt{was-introduced} \bullet b$ $,\ \lambda a.\lambda p.\lambda e.\ \mathbf{introduce}_{et}(e) \wedge \mathsf{PAT}(e,p)$ $\wedge \mathsf{AG}(e,a) \wedge$ $\exists g.\mathsf{GOAL}(e,g)\ \rangle$
intro_{to}	$\mathsf{np_{to}} \to \mathsf{np} \to \mathsf{vp}$	$\langle\ \lambda t.\lambda s.s \bullet \texttt{was-introduced} \bullet t$ $,\ \lambda g.\lambda p.\lambda e.\ \mathbf{introduce}_{et}(e) \wedge \mathsf{PAT}(e,p)$ $\wedge \exists a.\mathsf{AG}(e,a)$ $\wedge \mathsf{GOAL}(e,g)\ \rangle$
$\mathsf{intro}_{by,to}$	$\mathsf{np_{by}} \to \mathsf{np_{to}} \to \mathsf{np} \to \mathsf{vp}$	$\langle\ \lambda b.\lambda t.\lambda s.s \bullet \texttt{was-introduced} \bullet b \bullet t$ $,\ \lambda a.\lambda g.\lambda p.\lambda e.\ \mathbf{introduce}_{et}(e) \wedge \mathsf{PAT}(e,p)$ $\wedge \mathsf{AG}(e,a)$ $\wedge \mathsf{GOAL}(e,g)\ \rangle$

7 Conclusions

Q1: **How can optionality be incorporated to a formal semantic-syntactic framework?**

Optionality can be added to a grammatical framework by adding *option types* to the grammatical system. This addition consists of three basic rules: *option injection*, *universal filler introduc-*

tion, and *option analysis*. Using these extension, it is possible to mark arguments as optional.

Optional arguments can always be saturated with a special null argument (the universal filler) which carries no information or any appropriate ordinary argument (option injection). When a function has an optional type as its argument, the value coming from this optional argument might not carry any meaningful information. There are two cases: it might be the universal filler, or an ordinary value which is promoted to an optional value using option injection. To distinguish null values or option injected ordinary values, a type of case analysis is needed. The option operator takes an optional argument, a function to apply to provided arguments, and a default argument in case the optional argument is null. Depending on the case (filler or injected value), the option analysis rule selects the appropriate case.

How an optional value is dealt with is defined lexically, separate for each concrete component. In the morpho-syntactic component, missing optional values are usually resolved using a null string, and provided optional values are simply inserted at the appropriate position. In some cases, a missing value may trigger insertion of clitics, a change of morphology, or a change in word order. In the semantic component, an unfilled optional value can be resolved using a variety of methods: existential quantification, reflexivization, omission, which explain the entailments between a verb provided with different numbers of arguments.

This method of option types is in a sense too general: anything can be optional, and unfilled optional arguments can be resolved in any way. To impose restrictions on which components can be optional, and how missing optional arguments are resolved, we use *optionality markers* and *basic optionalization procedures*. Optionality markers are type annotations that specify that an argument should be optional in a certain way. Each marker has an associated basic optionalization procedure that specifies how an unfilled optional argument should be filled, for each concrete component. Using a recursive procedure, these basic optionalization procedures are generalized to a *general optionalization procedure*. The general optionalization procedure maps non-optional entries with optionality markers to optional entries, such that unfilled optional arguments will be resolved as determined by their markers.

For understood existential arguments this optionalization procedure is *general*: any number of arguments may be marked as optional. This elegantly covers cases such as passives of ditransitive verbs, which have multiple optional argument that when missing are interpreted as understood existentials. For other types of optional arguments, the procedure we proposed is only applicable for specific types of signs. Further research is needed to determine which of the other optionalization strategies we considered should and can be generalized.

Q2: **How can a verb with an optional argument compose with its argument when it is present?**

The option type extension includes the *option injection* rule. In combination with Hypothetical Reasoning any non-optional value can be made into a provided optional value, so that any allowing any suitable argument can be used as an optional argument.
These steps of composing a sign with an optional injected rule can be abbreviated, by placing a $\uparrow$ or $\downarrow$ marker on the optional argument and applying deoptionalization.

Q3: **How can an argument of a verb be missing and affect the interpretation of the verb?**

When an argument is missing it is filled using the *universal filler* $*$ which ensures that the default value of the option operator is returned. This default value can be anything, including an operation that saturates the unfilled slot of the optional argument using a variety of saturation procedures, like existential import, reflexivisation, reciprocalization. These saturation procedures give rise to the entailments between verbs supplied with different numbers of arguments.

Q4: **How can the narrow scope behavior of quantifiers introduced by implicit arguments be explained?**

Quantifiers introduced by missing arguments are introduced lexically, within the denotation of the verb, and hence have narrow scope with respect to quantifiers introduced elsewhere in the sentence.

Q5: **How should the syntactic and semantic differences between passives and unaccusatives be formalized?**

Both verbs in passive voice and unaccusatives with a transitive counterpart can be analyzed as having optional arguments. The behaviors of the optional arguments of these two constructions show some similarities. A passive verb takes an optional *by* phrase as an argument, and an unaccusative takes an optional agent argument. In both cases the agent argument is not obligatory but optional. The difference is that when the optional argument of a passive is missing the unfilled semantic slot is existentially saturated, whereas a missing agent of an unaccusative is saturated by a neutral element. With passives the absence of an optional argument will not reduce the number of roles involved with the event, with unaccusatives the agent role is omitted. The result of this is that missing arguments of passives give rise to existential equivalences while missing arguments of unaccusatives only lead to a one way entailment. Another difference is that a missing subject argument of an unaccusative causes a change in word order: the obligatory object takes the place of the missing subject. Finally, passive verbs can be derived from active verbs by a grammatical process, *passivisation*, while, at least in English, the derivation of the transitive counterpart of an unaccusative is a lexical process.

7.1 Further Research

- The framework presented here does not accommodate inflection. To keep the emphasis on arguments structure and optionality, the system is kept as simple as possible. Inflection is of course essential to a realistic grammatical framework. There are extensions to ACG and related systems to treat inflection (see [de Groote and Maarek, 2007, Ranta, 2004]). Modifications to the argument structure of verbs can affect case marking. It would be interesting to see if it is possible to accommodate these case markings in a such an extended framework with option types.
- Anaphoric implicit arguments cannot be analyzed in the given framework, because the semantics I used is static: the meaning of each sentence is analyzed in isolation. To accommodate anaphoric optional arguments, the semantics for the semantic component needs to be generalized to a dynamic semantics so that an analysis of verbs with anaphoric optional arguments can be given.
- I have not yet investigated the affects of adding option types and optional arguments to the complexity of a grammar, it would be

very interesting to know to what the effects of adding option types are on the difficulty of parsing and processing. I have demonstrated how an optional grammar can be mapped to a grammar without optionality, at the cost of lexical ambiguity. Which would be easier to parse?

References

[Blom et al., 2012] Blom, C., de Groote, P., Winter, Y., and Zwarts, J. (2012). Implicit arguments: Event modification or option type categories? In Aloni, M. et al., editors, *Proceedings of the Amsterdam Colloquium 2011, AC2011*, volume 7218 of *Lecture Notes in Computer Science, LNCS*, pages 240–250, Berlin/Heidelberg. Springer.

[Bresnan, 1978] Bresnan, J. (1978). A realistic transformational grammar. *Linguistic theory and psychological reality*, pages 1–59.

[Carlson, 1984] Carlson, G. (1984). Thematic roles and their role in semantic interpretation. *Linguistics*, 22:259–279.

[Chomsky, 1965] Chomsky, N. (1965). *Aspects of the Theory of Syntax*. The MIT press.

[Curry, 1961] Curry, H. (1961). Some logical aspects of grammatical structure. In *Structure of Language and its Mathematical Aspects: Proceedings of the Twelfth Symposium in Applied Mathematics*, pages 56–68.

[de Groote, 2001] de Groote, P. (2001). Towards abstract categorial grammars. In *Association for Computational Linguistics, 39th Annual Meeting and 10th Conference of the European Chapter, Proceedings of the Conference*, pages 148–155.

[de Groote and Maarek, 2007] de Groote, P. and Maarek, S. (2007). Type-theoretic extensions of Abstract Categorial Grammars. *Unpublished ms.*

[Dixon, 2000] Dixon, R. (2000). *Changing valency: Case studies in transitivity*. Cambridge Univ Pr.

[Dowty, 1982] Dowty, D. (1982). Grammatical Relations and Montague Grammar. *The nature of syntactic representation*, pages 79–130.

[Fodor and Fodor, 1980] Fodor, J. and Fodor, J. (1980). Functional structure, quantifiers, and meaning postulates. *Linguistic Inquiry*, 11(4):759–770.

[Kanazawa, 2007] Kanazawa, M. (2007). Parsing and generation as datalog queries. In *ANNUAL MEETING-ASSOCIATION FOR COMPUTATIONAL LINGUISTICS*, volume 45, page 176.

[Landau, 2010] Landau, I. (2010). The explicit syntax of implicit arguments. *Linguistic Inquiry*, 41(3):357–388.

[Landman, 2000] Landman, F. (2000). *Events and plurality: The Jerusalem lectures*, volume 76. Kluwer Academic Pub.

[Levin, 1993] Levin, B. (1993). *English verb classes and alternations: A preliminary investigation*, volume 348. University of Chicago press Chicago, IL.

[Mittwoch, 1982] Mittwoch, A. (1982). On the difference between eating and eating something: Activities versus accomplishments. *Linguistic Inquiry*, 13(1):pp. 113–122.

[Muskens, 2001] Muskens, R. (2001). Lambda grammars and the syntax-semantics interface. In *Proceedings of the Thirteenth Amsterdam Colloquium*, pages 150–155. Universiteit van Amsterdam.

[Ranta, 2004] Ranta, A. (2004). Grammatical framework. *J. Funct. Program.*, 14:145–189.

[Salvati, 2010] Salvati, S. (2010). On the membership problem for non-linear abstract categorial grammars. *Journal of Logic, Language and Information*, 19(2):163–183.

[Winter and Zwarts, 2011] Winter, Y. and Zwarts, J. (2011). Event semantics and Abstract Categorial Grammar. In Kanazawa, M. et al., editors, *Proceedings of Mathematics of Language, MOL12*, volume 6878 of *Lecture Notes in Artificial Intelligence, LNAI*, pages 174–191, Berlin. Springer-Verlag.

A The SIGNS interpreter

To learn more about ACG, to check the correctness of complex signs and fragments, and to typeset signs, I have developed a simple interpreter for ACG with option types, named SIGNS. This section will give a brief overview of the functionality of this tool and guide on how to define a fragment, type check it, and how to export signs to LaTex. A sign grammar is an ACG presented with the notation used in this thesis, as a list of signs: abstract constants together with their concrete terms.

A.1 Instructions

Defining a sign grammar

A sign grammar can be defined in a text file. It is defined by first giving the names of the abstract component and the concrete comp-ponenent, which is done as follows:

```
component Abstract = <Syntax,Semantic>
```

Next we define the basic type interpretation function, which defines how the abstract types are interpreted is given. This is simply a list associating each basic abstract type with its concrete types:

```
type_interpretations =
[ n      = <f , e -> t >
, np     = <f , e >
, s      = <f , t >
, vp     = <f , e -> t>
]
```

Next we can give a list of signs. The type of each constant has to be defined usign `::`, with the exception of strings (between double quotes), and the predefined constants `AG`,`PAT`,`GOAL`, and the logical operators. The syntax for lambda terms is standard, except that \ is used instead of λ.

```
JOHN :: np =
    < "John"
    , (John :: e)
    >
```

```
LIKES :: np -> np -> vp =
    < \o.\s.s + "likes" + o
    , \p.\a.\e.((Likes :: e -> t)(e)) /\ (AG e a) /\ (PAT e p)
    >
SOMEONE :: (np -> s) -> s =
    < \f.f("someone")
    , \f.exists x.f(x)
    >
READ :: np? -> np -> vp =
    < \o.\s.s + "read" + option(o,\o'.o',"")
    , \p.\a.\e.((Read :: e -> t)(e))  /\ (AG e a)
      /\ option(p,\p'.(PAT e p'),exists p'.PAT e p')
  >
```

Running SIGNS

The interpreter is started with the following commmand (provided that Haskell is installed):

```
runhaskell Signs.hs
```

Entering `:help` displays the available commands, with instructions on how to use them. Loading a grammar is done with the `:load` command, for example, to load the grammar used in this theses, enter:

```
 >:load opt
```

Now we can enter terms built out of the signs defined in `opt.signs`, which are then typed and interpreted. For example, the sign for the sentence Mary likes John:

```
 >LIKES JOHN MARY
```

This displays the typed sign for the sentence if it is well-typed, or gives an error message otherwise:

```
LIKES(JOHN)(MARY) :: vp =
    < "Mary" + "likes" + "John" :: f
    , \e.Likes(e) /\ AG(e,Mary) /\ PAT(e,John) :: et
    >
```

Signs can be pretty printed as Latex files using the `:savetex` command. The following command saves the Latex representation of the sign for the sentence Mary likes John to the file `sent1.tex`.

```
>:savetex sent1.tex LIKES JOHN MARY
```

This file can be included in a Latex file, it will be displayed as:

$$
\begin{array}{ll}
\mathsf{LIKES(JOHN)(MARY)} : \mathsf{vp} = & \\
\langle\ \texttt{Mary} \bullet \texttt{likes} \bullet \texttt{John} & : f \\
,\ \lambda \mathsf{e}.\mathbf{likes}_{et}(\mathsf{e}) \wedge \mathsf{AG}(\mathsf{e}, \mathbf{mary}_e) \wedge \mathsf{PAT}(\mathsf{e}, \mathbf{john}_e) & : et\ \rangle
\end{array}
$$

A.2 Implementation

Internally, the interpreter uses different formulation of ACG, based on the definitions given in [de Groote, 2001], as it is more suited for implementation. In that flavor of ACG, application, abstraction, reduction are defined over lambda terms instead of tuples of lambda terms. This makes it easier to use existing algorithms for reduction of λ-terms and type inference.
Below is the definition of ACG used for the implementation in the more traditional notation. There are some differences: the set of terms over a signature are non-linear λ-terms rather than linear and the addition of option types. I used non-linear lambda terms because certain linguistic phenomena which are relevant for the questions I wanted to answer, such as reflexivization, cannot be modeled using linear λ-terms. For the purposes of this thesis, I ignored some technical difficulties introduced by using non-linear λ-terms. See [de Groote and Maarek, 2007] for discussion and solutions to these issues.

B Option Types in ACG

Option types for traditional ACG's.

B.1 Signatures

A signature is a triple of a finite set of basic types A, a finite set of constants C and a typing function τ which assigns a type to each constant in C.

$\Sigma = \langle A, C, \tau \rangle$

B.2 Types

Let A be the set of basic types of a signature Σ. The set of types over signature Σ, $\mathcal{T}_\Sigma$ is defined inductively as:

if $\alpha \in A$ then $\alpha \in \mathcal{T}_\Sigma$	basic types
if $\alpha, \beta \in A$ then $(\alpha \to \beta) \in \mathcal{T}_\Sigma$	function types
if $a \in \mathcal{T}_\Sigma$ then $a^? \in \mathcal{T}_\Sigma$	option types

The only addition to the definition of the types is the unary $\cdot^?$ type constructor.

B.3 Terms

Let C be the set of constants of a signature Σ and V an infinite countable set of λ-variables. The set of λ-terms over C, Λ_Σ is defined inductively as:

if $c \in C$ then $c \in \Lambda_\Sigma$	constants
if $v \in V$ then $v \in \Lambda_\Sigma$	variables
if $x \in \Lambda_\Sigma$ and $v \in V$ then $\lambda v.x \in \Lambda_\Sigma$	lambda abstraction
if $m, n \in \Lambda_\Sigma$ then $m(n) \in \Lambda_\Sigma$	function application
$* \in \Lambda_\Sigma$	null option
if $x \in \Lambda_\Sigma$ then $\overline{x} \in \Lambda_\Sigma$	option injection
if $x, y, z \in \Lambda_\Sigma$ then $\mathsf{option}(x, y, z) \in \Lambda_\Sigma$	option analysis

Note that there are three additions compared to the standard definition of lambda terms: null option, option injection, and option analysis. Beside the usual reduction rules for lambda calculus, these additional reduction rules to handle option types:

$$\mathsf{option}(\overline{x}, f, d) \rightsquigarrow f(x)$$

$$\mathsf{option}(*, f, d) \rightsquigarrow d$$

B.4 Typing Rules

For each signature Σ, each term of that signature is associated with a type of that signature by these typing rules:

$$\Gamma \ \vdash_\Sigma \ c : \tau(c) \text{ (constant)}$$

$$\Gamma, v : \alpha \ \vdash_\Sigma \ v : \alpha \text{ (variable)}$$

$$\frac{\Gamma \ \vdash_\Sigma \ f : \alpha \to \beta \qquad \Delta \ \vdash_\Sigma \ x : \alpha}{\Gamma, \Delta \ \vdash_\Sigma \ f(x) : \beta} \text{ (application)}$$

$$\frac{\Gamma, v : \alpha \ \vdash_\Sigma \ m : \beta}{\Gamma \ \vdash_\Sigma \ \lambda v.m : \alpha \to \beta} \text{ (abstraction)}$$

$$\frac{\Gamma \ \vdash_\Sigma \ x : \alpha}{\Gamma \ \vdash_\Sigma \ \overline{x} : \alpha^?} \text{ (option injection)}$$

$$\frac{}{\Gamma \ \vdash_\Sigma \ * : \alpha^?} \text{ (universal filler)}$$

$$\frac{\Gamma \ \vdash_\Sigma \ x : \alpha^? \qquad \Gamma \ \vdash_\Sigma \ f : \alpha \to \beta \qquad \Gamma \ \vdash_\Sigma \ d : \beta}{\Gamma, \ \vdash_\Sigma \ \mathsf{option}(x, f, d) : \beta} \text{ (option elimination)}$$

B.5 Interpretation Functions

An interpretation function is defined as a pair $\langle F, G \rangle$, where

$$\begin{aligned} F &: \mathsf{basic\text{-}types}(A) \to \mathcal{T}_B \\ G &: \mathsf{constants}(A) \to \Lambda_B \end{aligned}$$

An interpretation function defines how to map constants and atomic types from signature A to terms and types of signature B.
By taking the unique homomorphic extensions of F and G, we get a map from terms and types of signature a to terms and types of signature B.

The unique homomorphic extension of F, $\hat{F}$ is defined as:

$$\begin{aligned} \hat{F}(c) &= F(c) \\ \hat{F}(v) &= v \\ \hat{F}(m(n)) &= (\hat{F}(m))(\hat{F}(n)) \\ \hat{F}(*) &= * \\ \hat{F}(\overline{x}) &= \overline{\hat{F}(x)} \\ \hat{F}(\mathsf{option}(x, f, d) &= \mathsf{option}(\hat{F}(x), \hat{F}(f), \hat{F}(d)) \end{aligned}$$

The unique homomorphic extension of G, $\hat{G}$ is defined as :

$$\begin{aligned}
\hat{G}(a) &= G(a)\\
\hat{G}(\alpha \to \beta) &= \hat{G}(\alpha) \to \hat{G}(\beta)\\
\hat{G}(\alpha^?) &= \hat{G}(\alpha)^?
\end{aligned}$$