Type theory and language
From perception to linguistic communication

Robin Cooper

Draft, June 17, 2014

Contents

Acknowledgements

I am grateful to many people for discussion which has led to significant changes in this material. Among them are: Simon Dobnik, Tim Fernando, Staffan Larsson, Bengt Nordström, Aarne Ranta. None of these people is responsible for what I have done with their ideas and suggestions.

This research was supported in part by the following projects: Records, types and computational dialogue semantics, Vetenskapsrådet, 2002-4879, Library-based grammar engineering, Vetenskapsrådet, 2005-4211, and Semantic analysis of interaction and coordination in dialogue (SAICD), Vetenskapsrådet, 2009-1569.

Chapter 1

From perception to intensionality

1.1 Perception as type assignment

Kim is out for a walk in the park and sees a tree. She knows that it is a tree immediately and does not really have to think anything particularly linguistic, such as "Aha, that's a tree". As a human being with normal visual perception, Kim is pretty good at recognizing something as a tree when she sees it, provided that it is a fairly standard exemplar, and the conditions are right: for example, there is enough light and she is not too far away or too close. We shall say that Kim's perception of a certain object, a, as a tree involves the ascription of a type *Tree* to a. In terms of modern type theory (as in Martin-Löf (1984); Nordström *et al.* (1990)), we might say that Kim has made the *judgement* that a is of type *Tree* (in symbols a : *Tree*).

Objects can be of several types. An object a can be of type *Tree* but also of type *Oak* (a subtype of *Tree*, since all objects of type *Oak* are also of type *Tree*) and *Physical Object* (a supertype of *Tree*, since all objects of type *Tree* are of type *Physical Object*). It might also be of an intuitively more complicated type like *Objects Perceived by Kim* which is neither a subtype nor a supertype of *Tree* since not all objects perceived by Kim are trees and not all trees are perceived by Kim.

There is no perception without some kind of judgement with respect to types of the perceived object. When we say that we do not know what an object is, this normally means that we do not have a type for the object which is narrow enough for the purposes at hand. I trip over something in the dark, exclaiming "What's that?", but my painful physical interaction with it through my big toe tells me at least that it is a physical object, sufficiently hard and heavy to offer resistance to my toe. The act of perceiving an object is perceiving it *as* something. You cannot perceive something without ascribing some type to it, even if it is a very general type such as *thing* or *entity*.

Recognizing something as a tree may be immediate and not involve conscious reasoning. Recognizing a tree as an aspen, an elm or a tree with Dutch elm disease may involve closer inspection and some conscious reasoning about the shape of the leaves or the state of the bark. For humans the relating of objects to certain types can be the result of a long chain of reasoning involving a great deal of conscious effort. But whether the perception is immediate and automatic or the result of a conscious reasoning process, from a logical point of view it still seems to involve the ascription of a type to an object.

The kind of types we are talking about here correspond to pretty much any useful way of classifying things and they correspond to what might be called properties in other theories. For example, in the classical approach to formal semantics developed by Montague (1974) and explicated by Dowty *et al.* (1981) among many others, properties are regarded not as types but as functions from possible worlds and times to (the characteristic functions of) sets of entities, that is, the property *tree* would be a function from possible worlds and times to the set of all entities which are trees at that world and time. Montague has types based on a version of Russell's (1903) simple theory of types but they were "abstract" types like *Entity* and *Truth Value* and types of functions based on these types rather than "contentful" types like *Tree*. Type theory for Montague was a way of providing basic mathematical structure to the semantic system in a way that would allow the generation of interpretations of infinitely many natural language expressions in an orderly fashion that would not get into problems with logical paradoxes. The development of type theory which we will undertake here can be regarded as an enrichment of an "abstract" type theory like Montague's with "contentful" types. We want to do this in a way that allows the types to account for content and relate to cognitive processing such as perception. We want our types to have psychological relevance and to correspond to what Gibson (1986) might call *invariants*, that is, aspects that we can perceive to be the same when confronted with similar objects or the same object from a different perspective. In this respect our types are similar to notions developed in situation theory and situation semantics (Barwise and Perry, 1983; Barwise, 1989).

Gibson's notion of attunement is adopted by Barwise and Perry. The idea is that certain organisms are attuned to certain invariants while others are not. Suppose that Kim perceives a cherry tree with flowers and that a bee alights on one of the flowers. One assumes that the bee's experience of the tree is very different from Kim's. It seems unlikely that the bee perceives the tree as a tree in the sense that Kim does and it is not at all obvious that the bee perceives the tree in its totality as an object. Different species are attuned to different types and even within a species different individuals may vary in the types to which they are attuned. This means that our perception is limited by our cognitive apparatus – not a very surprising fact, of course, but philosophically very important. If perception involves the assignment of types to objects and we are only able to perceive in terms of those types to which we are attuned, then as Kant (1781) pointed out we are not actually able to be aware of *das Ding an sich* ("the thing itself"), that is, we are not able to be aware of an object independently of the categories (or types) which are available to us through our cognitive apparatus.

1.2 Modelling type systems in terms of mathematical objects

In order to make our theory precise we are going to create models of the systems we propose as mathematical objects. This represents one of the two main strategies that have been employed in logic to create rigorous theories. The other approach is to create a formal language to describe the objects in the theory and define rigorous rules of inference which explicate the properties of the objects and the relations that hold between them. At a certain level of abstraction the two approaches are doing the same thing – in order to characterize a theory you need to say what objects are involved in the theory, which important properties they have and what relations they enter into. However, the two approaches tend to get associated with two different logical traditions: the *model theoretic* and *proof theoretic* traditions.

The philosophical foundation of type theory (as presented, for example, by Martin-Löf (1984)) is normally seen as related to intuitionism and constructive mathematics. It is, at bottom, a proof-theoretic discipline rather than a model-theoretic one (despite the fact that model theories have been provided for some type theories). However, it seems that many of the ideas in type theory that are important for the analysis of natural language can be adopted into the classical set theoretic framework familiar to linguists from the classical model-theoretic canon of formal semantics starting from Montague (1974).

Your theory is not very interesting if it does not make *predictions*, that is, by making certain assumptions you can infer some conclusions. This gives you one way to test your theory: see what you can conclude from premises that you know or believe to be true and then test whether the conclusion is actually true. If you can show that you theory allows you to predict some conclusion and its negation, then your theory is *inconsistent*, which means that it is not useful as a scientific theory. One way to discover whether a theory is consistent or not is to formulate it very carefully and explicitly so that you can show mathematical properties of the system and any inconsistencies will appear.

From the informal discussion of type theory that we have seen so far it is clear that it should involve two kinds of entity: the types and the objects which are of those types. (Here we use the word "entity" not in the sense that Montague did, that is, basic individuals, but as an informal notion which can include both objects and types.) This means that we should characterize a type theory with two domains: one domain for the objects of the types and another domain for the types to which these objects belong. Thus we see types as theoretical entities in their own right, not, for example, as collections of objects. Diagrammatically we can represent this as in Figure 1.1 where object a is of type T_1.

A system of basic types consists of a set of types which are *basic* in the sense that they are not analyzed as *complex* objects composed of other objects in the theory. Each of these types is associated with a set of objects, that is, the objects which are of the type, that is the *witnesses* for the type. Thus if T is a type and A is the set of objects associated with T, then a is of type

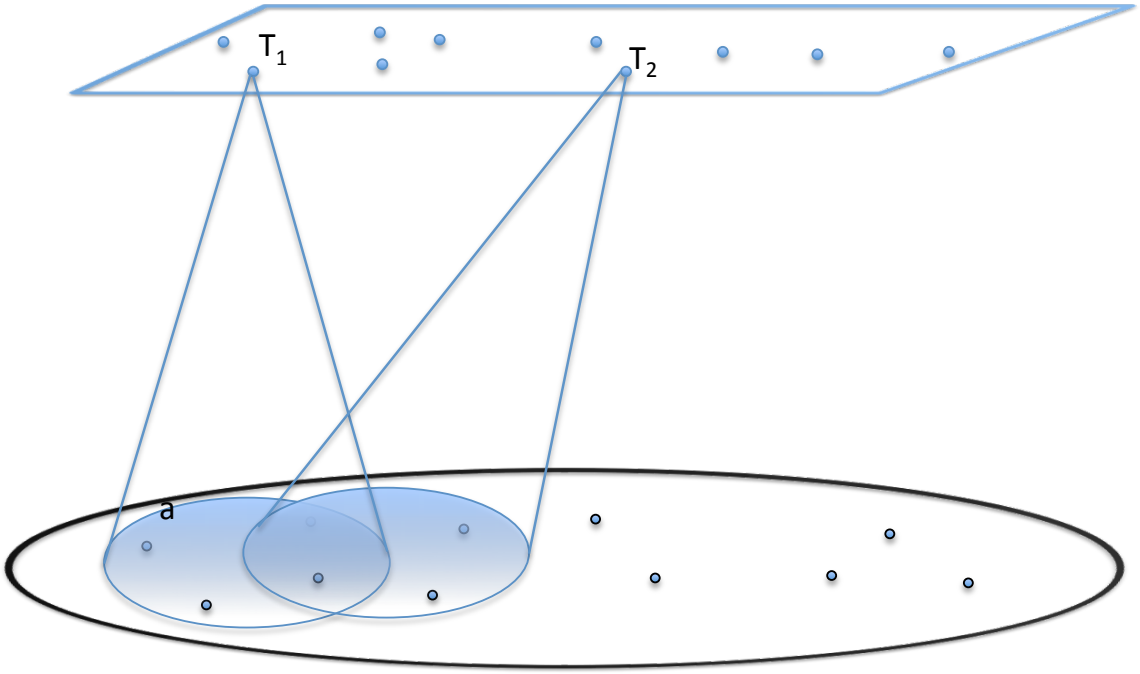


Figure 1.1: System of basic types

T (in symbols, $a : T$) just in case $a \in A$. We require that any object a which is a witness for a basic type is not itself one of the types in the system. A type may be *empty* in the sense that it is associated with the empty set, that is, there is nothing of that type.

Notice that we are starting with the types and associating sets of objects with them. This means that while there can be types for which there are no witnesses, there cannot be objects which do not belong to a type. This relates back to our claim in Section 1.1 that we cannot perceive an object without assigning a type to it.

Notice also that the sets of objects associated with types may have members in common. Thus it is possible for objects to belong to more than one type. This is important if we want to have basic types *Elm*, *Tree* and *Physical Object* and say that a single object a belongs to all three types as discussed in Section 1.1.

An extremely important property of this kind of type system is that there is nothing which prevents two types from being associated with exactly the same set of objects. In standard set theory the notion of set is *extensional*, that is sets are defined by their membership. You cannot have two distinct sets with the same members. The choice of defining types as entities in their own right rather than as the sets of their witnesses, means that they can be *intensional*, that is, you can have

more than one type with the same set of witnesses. This can be important for the analysis of natural language words like *groundhog* and *woodchuck* which (as I have learned from the literature on natural language semantics) are the same animal. In this case one may wish to say that you have two different words which correspond to the same type, rather than two types with the same *extension* (that is, set of witnesses). Such an analysis is less appealing in the case of *unicorn* and *centaur*, both mythical animals corresponding to types which have an empty extension. If types were extensional, there would only be one empty type (just as there is only one empty set in set theory). In the kind of possible world semantics espoused by Montague the distinction between *unicorn* and *centaur* was made by considering their extension not only in the actual world (where both are empty) but also in all possible worlds, since there will be some worlds in which the extensions are not the same. However, this kind of possible worlds analysis of intensionality fails when you have types whose extensions cannot possibly be different. Consider *round square* and *positive number equal to* $2 - 5$. The possible worlds analysis cannot distinguish between these since their extensions are both empty no matter which possible world you look at.

Finally, notice that there may be different systems of basic types, possibly with different types and different objects. One way of exploiting this would be to associate different systems with different organisms as discussed in Section 1.1. (Below we will see different uses of this for the analysis of types which model the cognitive system of a single agent.) Thus properly we should say that an object a is of type T with respect to a basic systems of types $\mathbf{TYPE}_B$, in symbols, $a :_{\mathbf{TYPE_B}} T$. However, we will continue to write $a : T$ in our informal discussion when there is no danger of confusion.

The definition of a system of basic types is made precise in Appendix A.2.

What counts as an object may vary from agent to agent (particularly if agents are of different species). Different agents have what Barwise (1989) would call different *schemes of individuation*. There appears to be a complex relationship between the types that an agent is attuned to and the parts of the world which the agent will perceive as an object. We model this in part by allowing different type systems to have different objects. In addition we will make extensive use in our systems of a basic type *Ind* for "individual" which corresponds to Montague's notion of "entity". The type *Ind* might be thought of as modelling a large part of an agent's scheme of individuation in Barwise's sense. However, this clearly still leaves a great deal to be explained and we do this in the hope that exploring the nature of the type systems involved will ultimately give us more insight into how individuation is achieved.

1.3 Situation types

Kim continues her walk in the park. She sees a boy playing with a dog and notices that the boy gives the dog a hug. In perceiving this event she is aware that two individuals are involved and

that there is a relation holding between them, namely hugging. She also perceives that the boy is hugging the dog and not the other way around. She sees that a certain action (hugging) is being performed by an agent (the boy) on a patient (the dog). This perception seems more complex than the classification of an individual object as a tree in the sense that it involves two individual participants and a relation between them as well as the roles those two individuals play in the relation. While it is undoubtably more complex than the simple classification of an object as a tree, we want to say that it is still the assignment of a type to an object. The object is now an event and she classifies the event as a hugging event with the boy as agent and the dog as patient. We shall have complex types which can be assigned to such events.

Complex types are constructed out of other entities in the theory. As we have just seen, cognitive agents, in addition to being able to assign types to individual objects like trees, also perceive the world in terms of states and events where objects have properties and stand in relations to each other – what Davidson (1967) called events and Barwise and Perry (1983) called situations. We introduce types which are constructed from predicates (like 'hug') and objects which are arguments to this predicate like a and b. We will represent such a constructed type as hug(a,b) and we will sometimes call it a *ptype* to indicate that it is a type whose main constructor is a predicate. What would an object belonging to such a type be? According to the type-theoretic approach introduced by Martin-Löf it should be an object which constitutes a proof that a is hugging b. For Martin-Löf, who was considering mathematical predicates, such proof objects might be numbers with certain properties, ordered pairs and so on. Ranta (1994) points out that for non-mathematical predicates the objects could be events as conceived by Davidson (1967, 1980). Thus hug(a,b) can be considered to be an event or a situation type. In some versions of situation theory Barwise (1989); Seligman and Moss (1997), objects (called *infons*) constructed from a relation and its arguments was considered to be one kind of situation type. Thus one view would be that ptypes are playing a similar role in type theory to the role that infons play in situation theory.

What kind of entity are predicates? The notion is made precise in Appendix A.3.1. The important thing about predicates is that they come along with an *arity*. The arity of a predicate tells you what kind of arguments the predicate takes and what order they come in. For us the arity of a predicate will be a sequence of types. The predicate 'hug' as discussed above we can think of as a two-place predicate both of whose arguments must be of type *Ind*, that is, an individual. Thus the arity of 'hug' will be $\langle$*Ind*, *Ind*$\rangle$. The idea is that if you combine a predicate with arguments of the appropriate types in the appropriate order indicated by the arity then you will have a type. Thus if a : *Ind* and b : *Ind* then hug(a,b) will be a type, intuitively the type of situation where a hugs b.

It may be desirable to allow some predicates to combine with more than one assortment of argument types. Thus, for example, one might wish to say that the predicate 'believe' can combine with two individuals just like 'hug' (as in *Kim believes Sam*) or with an individual and a "proposition" (as in *Kim believes that Sam is telling the truth*). Similarly the predicate 'want' might be both a two-place predicate for individuals (as in *Kim wants the tree*) or a two-place predicate

between individuals and "properties" (as in *Kim wants to own the tree*). We shall have more to say about "propositions" and "properties" later. For now, we just note that we want to allow for the possibilities that predicates can be *polymorphic* in the sense that there may be more than one sequence of types which characterize the arguments they are allowed to combine with. The sequences need not even be of the same length (consider *Kim walked* and *Kim walked the dog*). We thus allow for the possibility that these pairs of natural language examples can be treated using the same polymorphic predicate. Another possibility, of course, is to say that the English verbs can correspond to different (though related) predicates in the example pairs and not allow this kind of predicate polymorphism in the type theory. We do not take a stand on this issue but merely note that both possibilities are available. If predicates are to be considered polymorphic then the arity of a predicate can be considered to be a set of sequences of types.

Predicates can be considered as functions from sequences of objects matching their arity to types. As such they would be a *dependent type*, that is, an entity which returns a type when provided with an appropriate object or sequence of objects. However, we have not made this explicit in Appendix A.3.1.

A system of complex types (made precise in Appendix A.3.2) adds to a system of basic types a collection of types constructed from a set of predicates with their arities, that is, it adds all the types which you can construct from the predicates by combining them with objects of the types corresponding to their arities according to the types in the rest of the system. The system also assigns a set of objects to all the types thus constructed from predicates. Many of these types will be assigned the empty set. Intuitively, if we have a type hug(c,d) and there are no situations in which c hugs d then there will be nothing in the extension of hug(c,d), that is, it will be assigned the empty set in the system of complex types. Notice that the intensionality of our type system becomes very important here. There may be many individuals x and y for which hug(x,y) is empty but still we would want to say that the types resulting from the combination of 'hug' with the various different individuals corresponds to different types of situations. There are thus two important functions in a system of complex types: one, which we call A, which comes from the system of basic types embedded in the system and assigns extensions to basic types and the other, which we call F, which assigns extensions to types constructed from predicates and arguments corresponding to the arity of the predicates. We have chosen the letters A and F because they are used very often in the characterization of models of first order logic. A model for first order logic is often characterized as a pair $\langle A, F\rangle$ where A is the domain and F a function which assigns denotations to the basic expressions (constants and predicates) of the logic. In a slight variation on classical first order logic A may be a sorted domain, that is the domain is not a single set but a set divided into various subsets, corresponding to *sorts*. For us, A characterizes assignments to basic types and thus provides something like a sorted domain in first order model theory. In first order logic F gives us what we need to know to determine the truth of expressions like hug(a,b) in first order logic. Thus F will assign to the predicate 'hug' a set of ordered pairs telling us who hugs whom. Our F also give us the information we need in order to tell who stands in a predicate relation. However, it does this, not by assigning a set of ordered n-tuples to each predicate, but by assigning sets of witnesses (or "proofs") to each type constructed from

a predicate with appropriate arguments. The set of ordered pairs assigned to 'hug' by the first order logic F corresponds to the set of pairs of arguments $\langle x, y\rangle$ for which the F in a complex system of types assigns a non-empty set. For this reason we call the pair $\langle A, F\rangle$ a *model* within the type system, even though it is not technically a model in the sense of model theory for logic. The correspondence will become important below, however.

Kim sees this situation where a (the boy) hugs b (the dog) and perceives it to be of type hug(a,b). However, there are intuitively other types which she could assign to this situation other than the type of situation where a hugs b which is represented here. For example, a more general type, which would be useful in characterizing all situations where hugging is going on between any individuals, is that of "situation where one individual hugs another individual". Another type of situation she might use is that of "situation where a boy hugs a dog". This is a more specific type than "situation where one individual hugs another individual" but still does not tie us down to the specific individuals a and b as hug(a,b) does.

There are at least two different ways in type theory to approach these more general types. One is to use Σ-*types* such as (1).

(1) a. Σx:*Ind*.Σy:*Ind*.hug(x,y)

b. Σx:*Boy*.Σy:*Dog*.hug(x,y)

In general $\Sigma x{:}T_1.T_2(x))$ will have as witnesses any ordered pair the first member of which is a witness for T_1 and the second member of which is a witness for $T_2(x)$. Thus this type will be non-empty ("true") just in case there is something a of type T_1 such that there is something of type $T_2(a)$. This means that Σ-types correspond to existential quantification. A witness for (1a) would be $\langle a, \langle b, s\rangle\rangle$ where a:*Ind*, b:*Ind* and s:hug(a,b). If there is such a witness then some individual hugs another individual and conversely if some individual hugs another individual there will be a witness for this type. Σ-types are exploited for the semantics of natural language by Ranta (1994) among others.

Another approach to these more general types is to use *record types* such as (2).

(2) a. $\begin{bmatrix} \text{x} & : & \textit{Ind} \\ \text{y} & : & \textit{Ind} \\ \text{c} & : & \text{hug(x,y)} \end{bmatrix}$

b. $\begin{bmatrix} \text{x} & : & \textit{Boy} \\ \text{y} & : & \textit{Dog} \\ \text{c} & : & \text{hug(x,y)} \end{bmatrix}$

We make the notion of record type precise in Appendix A.11. Record types consist of sets of fields such as $\left[\text{x:}\mathit{Ind}\right]$ and $\left[\text{c:hug(x,y)}\right]$. Fields themselves are pairs consisting of a *label* such as 'x' or 'c' in the first position (before the ':' in our notation) and a type in the second position. You cannot have more than one field with the same label in a record type. The witnesses of record types are *records*. These are also sets of fields, but in this case the fields consist of a label and an object belonging to a type. A record, r, belongs to a record type, T, just in case r contains fields with the same labels as those in T and the objects in the fields in r are of the type with the corresponding label in T. The record may contain additional fields with labels not mentioned in the record type with the restriction there can only be one field within the record with a particular label. Thus both (3a) and (3b) are records of type (2a).

(3)

a. $\begin{bmatrix} \text{x} & = & a \\ \text{y} & = & b \\ \text{c} & = & s \end{bmatrix}$ where a:*Ind*, b:*Ind* and s:hug(a,b)

b. $\begin{bmatrix} \text{x} & = & d \\ \text{y} & = & e \\ \text{c} & = & s' \\ \text{z} & = & f \\ \text{w} & = & g \end{bmatrix}$ where d:*Ind*, e:*Ind*, s':hug(d,e) and f and g are objects of some type

Note that in our notation for records we have '=' between the two elements of the field whereas in record types we have ':'. Note also that when we have types constructed from predicates in our record types and the arguments are represented as labels as in (2a) this means that the type is *dependent* on what objects you choose for those labels in the object of the record type. Thus in (3a) the type of the object labelled 'c' is hug(a,b) whereas in (3b) the type is hug(d,e). Actually, the notation we are using here for the dependent types is a convenient simplification of what is needed as we explain in Appendix A.11.

Record types and Σ-types are very similar in an important respect. The type (2a) will be non-empty ("true") just in case there are individuals x and y such that x hugs y. Thus both record types and Σ-types can be used to model existential quantification. In fact record types and Σ-types are so similar that you would probably not want to have both kinds of types in a single system and we will not use Σ-types. We have chosen to use record types for a number of reasons:

fields are unordered The Σ-types in (4) are distinct, although there is an obvious equivalence which holds between them.

(4) a. Σx:*Ind*.Σy:*Ind*.hug(x,y)

b. Σy:*Ind*.Σx:*Ind*.hug(x,y)

They are not only distinct types but they also have distinct sets of witnesses. The object $\langle a, \langle b, s\rangle\rangle$ will be of type (4a) just in case $\langle b, \langle a, s\rangle\rangle$ is of type (4b). In contrast, since we are regarding record types (and records) as *sets* of fields, (5a,b) are variant notations for the same type.

(5) a. $\begin{bmatrix} \text{x} & : & \textit{Ind} \\ \text{y} & : & \textit{Ind} \\ \text{c} & : & \text{hug(x,y)} \end{bmatrix}$

b. $\begin{bmatrix} \text{y} & : & \textit{Ind} \\ \text{x} & : & \textit{Ind} \\ \text{c} & : & \text{hug(x,y)} \end{bmatrix}$

labels Record types (and their witnesses) include labelled fields which can be used to access "components" of what is being modelled. This is useful, for example, when we want to analyze anaphoric phenomena in language where pronouns and other words refer back to parts of previous meanings in the discourse. They can also be exploited in other cases where we want to refer to "components" of utterances or their meanings as in clarification questions.

discourse representation The labels in record types can play the role of discourse referents in discourse representation structures (DRSs, Kamp and Reyle, 1993) and record types of the kind we are proposing can be used to model DRSs.

dialogue game boards Record types have been exploited to model dialogue game boards or information states (see in particular Ginzburg, 2012).

feature structures Record types can be used to model the kind of feature structures that linguists like to use (as, for example, in linguistic theories like Head Driven Phrase Structure Grammar, HPSG, Sag *et al.*, 2003). Here the labels in record types correspond to attributes in feature structures.

frames Record types can also be used to model something very like the kinds of frames discussed in frame semantics (Fillmore, 1982, 1985; Ruppenhofer *et al.*, 2006). Here the labels in record types correspond to roles (frame elements).

For discussion of some of the various uses to which record types can be put see Cooper (2005). We will take up all of the uses named here as we progress.

Another way of approaching these more general types in type theory is to use *contexts*. In (6) we take *True* to be the type of non-empty types.

(6) a. x : *Ind*, y : *Ind* $\vdash$ hug(x,y) : *True*

b. x : *Boy*, y : *Dog* $\vdash$ hug(x,y) : *True*

(6a,b) mean in a context where x and y are individuals or a boy and a dog respectively the type hug(x,y) is non-empty. This notation is normally taken to mean universal quantification over the parameters or variables in the context (i.e. sequence of parametric type judgements) to the left of '$\vdash$'. Thus they would mean that for any two individuals or pair of a boy and a dog, the first hugs the second. However, we can also devise ways for thinking of existential quantification over the variables of the context, e.g. for some boy, x, and some dog, y, the type hug(x,y) is non-empty. We can also think of the contexts as being objects belonging to types in our type theory. Records and record types give us a way of doing this. Thus, for example, (2a) models the type of context which might be represented as the sequence of parametric type judgements given in (7).

(7) x : *Ind*, y : *Ind*, c : hug(x,y)

As in the comparison with Σ-types there is a difference in that the judgements in a standard type theory context are ordered whereas the fields in a record type are unordered. This means that technically (8) is a distinct context from (7) even though there is an obvious equivalence between them.

(8) y : *Ind*, x : *Ind*, c : hug(x,y)

They correspond to the same record type, however. Since we will use record types to model type theoretic contexts and records to model instantiations of contexts we will not introduce a separate notion of context.

Thus we use record types to replace both the Σ-types and contexts that one often finds in standard versions of type theory.

The introduction of predicates and ptypes raises some new questions. We said above that the arity of 'hug' is ⟨*Ind*,*Ind*⟩. However, when we look at (2b) where the types labelled with 'x' and 'y' are *Boy* and *Dog* we see that there is nothing explicit here that requires that the two arguments of 'hug' are of type *Ind*. One obvious way to achieve this would be to require that *Boy* and *Dog* are subtypes of *Ind*, that is, that any object of type *Boy* is also of type *Ind* and similarly for *Dog*. However, now that we have introduced predicates there is nothing to stop us having two predicates 'boy' and 'dog' with arity ⟨*Ind*⟩. Thus we could have the record type (9).

(9)
$$\begin{bmatrix} x & : & \mathit{Ind} \\ c_{boy} & : & boy(x) \\ y & : & \mathit{Ind} \\ c_{dog} & : & dog(y) \\ c_{hug} & : & hug(x,y) \end{bmatrix}$$

How do we choose between a type like (9) where common nouns like *boy* and *dog* correspond to one-place predicates and a type like (2b) where common nouns correspond to basic types? One advantage is that (9) explicitly represents that the arity of 'hug' is fulfilled. Another advantage is that many, and on some analyses possibly all, nouns in natural languages will in more detailed treatments correspond to predicates of more than one argument. Consider, for example, the fact that boys grow into men. The same individual can be a boy at one time and a man at a later time. One way of treating this is to say that 'boy' is a predicate of two arguments with arity ⟨*Ind*,*Time*⟩. In fact if we are going to deal with tense and aspect in natural language in this way we will probably want to add time arguments to most if not all of our predicates and thus allow ourselves record types like (10).

(10)
$$\begin{bmatrix} \text{e-time} & : & \mathit{Time} \\ x & : & \mathit{Ind} \\ c_{boy} & : & boy(x,\text{e-time}) \\ y & : & \mathit{Ind} \\ c_{dog} & : & dog(y,\text{e-time}) \\ c_{hug} & : & hug(x,y,\text{e-time}) \end{bmatrix}$$

where 'e-time' stands for "event time". Here we have required that the times in all the predicate fields be the event time but this is not always the case. Consider (11).

(11) The minister smoked pot in his youth

Here the time of the pot-smoking event most likely precedes the time of the pot-smoking individual being a minister. We will thus use our basic types for basic ontological categories like

individual and *time* and use predicates for words that occur in natural language. Predicates can be n-ary whereas our types will always be unary. Note that a ptype like hug(a,b,t) is constructed from a ternary predicate 'hug' but the type itself is a unary type of situations. Thus we might have the judgement s : hug(a,b,t).

Below we will propose an alternative to this treatment of time as an argument. There is, however, another reason for allowing predicates corresponding to nouns to have more than one argument. This is the existence of relational nouns such as *friend* or *daughter*. (See Partee and Borschev, 2012 for recent discussion.)

In this book we will reserve basic types for two kinds of types: (i) those which correspond to intuitively fundamental ontological categories such as individual and (ii) those types which require a recursive definition to characterize the set of their witnesses. The latter is for a technical reason: defining recursive types as, for instance, record types could lead to the types themselves being a non-well-founded set of ordered pairs which contain themselves. We will discuss this more (????) when recursive types become relevant.

1.4 The string theory of events

Kim stands and watches the boy and the dog for a while. They start to play fetch.[1] This is a moderately complex game in that it consists of a number of components which are carried out in a certain order. The boy picks up a stick, attracts the attention of the dog (possibly shouting "Fetch!"), and throws the stick. The dog runs after the stick, picks it up in his mouth and brings it back to the boy. This sequence can be repeated arbitrarily many times. One thing that becomes clear from this is that events do not happen in a single moment but rather they are stretched out over intervals of time, characterized by the sub-events that constitute them. So if we were to have a type of event (that is, a kind of situation) play_fetch(a,b,c) where a is a human, b is a dog and c is a stickwe can say something about the series of subevents that we have identified. So we might draw an informal diagram something like Figure 1.2.

In an important series of papers including Fernando (2004, 2006, 2008, 2009, 2011), Fernando introduces a finite state approach to event analysis where events are analyzed in terms of finite state automata something like what we have represented in Figure 1.3. Such an automaton will recognize a string of sub-events. The idea is that our perception of complex events can be seen as strings of punctual observations similar to the kind of sampling we are familiar with from audio technology and digitization processing in speech recognition. Thus events can be analyzed as strings of smaller events. What we mean by a string is made precise in Appendix A.13. Any object of any type can be part of a string. Any two objects (including strings themselves), s_1

[1] `http://en.wikipedia.org/wiki/Fetch_(game)`, accessed 10th Oct 2011.

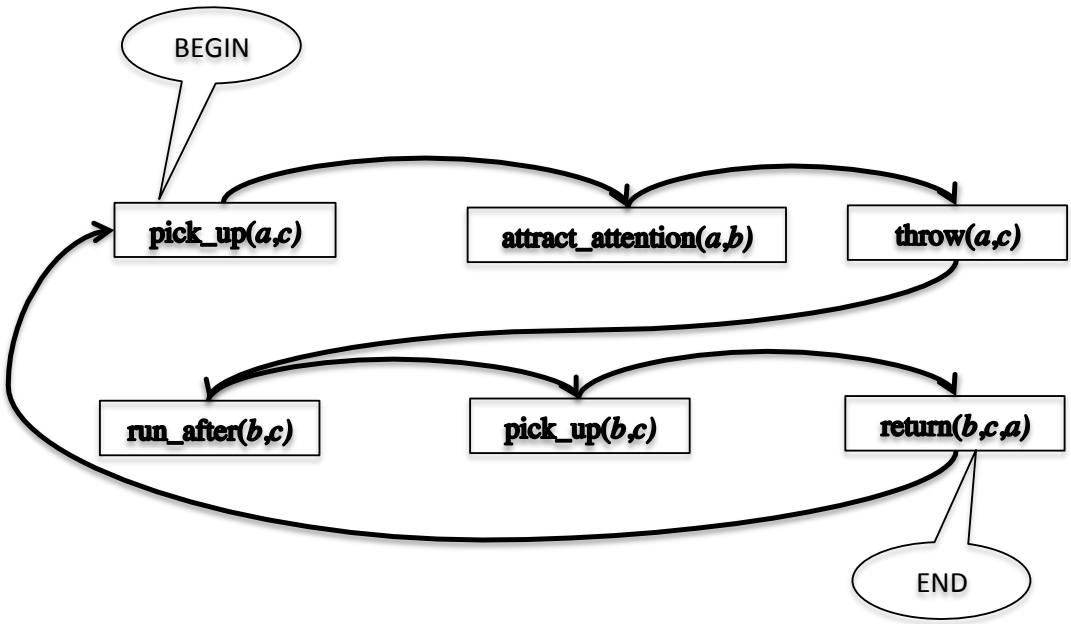


Figure 1.2: play_fetch(a,b,c)

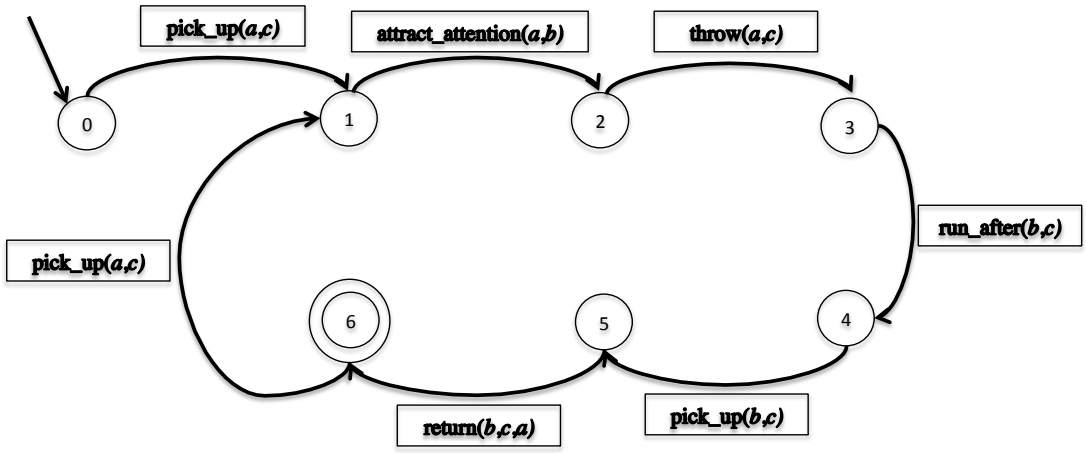


Figure 1.3: play_fetch(a,b,c) as a finite state machine

and s_2, can be *concatenated* to form a string $s_1{}^\frown s_2$. An important property of concatenation is *associativity*, that is if we concatenate s_1 with s_2 and then concatenate the result with s_3 we get the same string that we would obtain by concatenating s_2 with s_3 and then concatenating s_1 with the result. In symbols: $(s_1{}^\frown s_2){}^\frown s_3 = s_1{}^\frown(s_2{}^\frown s_3)$. For this reason we normally write $s_1{}^\frown s_2{}^\frown s_3$ (without the parentheses) or simply $s_1 s_2 s_3$ if it is clear from the context that we mean this to be string concatenation. Following Fernando we will use these strings to give us our notion of temporal order.

Although we will present strings in this way, we will model them as records with distinguished labels related to the natural numbers, t_0, t_1, ... ('t' for "time"). The field labelled t_n will correspond to the nth place in the string. Thus a string of objects $a_1 a_2 a_3$ will be the record in (12).

(12)
$$\begin{bmatrix} \mathrm{t}_0 & = & a_1 \\ \mathrm{t}_1 & = & a_2 \\ \mathrm{t}_2 & = & a_3 \end{bmatrix}$$

The concatenation of (12) with the string a_4, that is, (13a), will be (13b).

(13) a.
$$\begin{bmatrix} \mathrm{t}_0 & = & a_4 \end{bmatrix}$$

b.
$$\begin{bmatrix} \mathrm{t}_0 & = & a_1 \\ \mathrm{t}_1 & = & a_2 \\ \mathrm{t}_2 & = & a_3 \\ \mathrm{t}_3 & = & a_4 \end{bmatrix}$$

We will continue to represent strings for convenience in the traditional way but modelling strings as records will become important when following paths in records down to elements in strings. We will use $s[n]$ to represent the nth element in a string s. But in terms of the record notation this is just a convenient abbreviation for $s.\mathrm{t}_n$.

Now let us build further on the types that we have introduced so far to include string types. For any two types, T_1 and T_2, we can form the type $T_1{}^\frown T_2$. This is the type of strings $a{}^\frown b$ where $a : T_1$ and $b : T_2$. The concatenation operation on types (just like that on objects) is associative so we do not use parentheses when more than one type is involved, e.g. $T_1{}^\frown T_2{}^\frown T_3$.

Let us return to Kim watching the boy, a, playing fetch with the dog, b, using the stick, c. She perceives the event as being of type play_fetch(a,b,c). But what does it mean to be an event of this type? Given our concatenation types we can build a type which corresponds to most of what we have sketched in Figure 1.2, namely (14).

(14) pick_up(a,c)⌢attract_attention(a,b)⌢throw(a,c)⌢run_after(b,c)⌢pick_up(b,c)⌢return(b,c,a)

(14) is a type corresponding to everything we have represented in Figure 1.2 except for the arrow which loops back from the end state to the start state. In order to get the loop into the event type we will use a kind of type which introduces a Kleene-+. In standard notations for strings s^+ stands for a string consisting of one or more occurrences of s.[2] We will adopt this into types by saying that for any type T there is also a type T^+ which is the type of strings of objects of type T containing one or more members. (See Appendix A.13 for a more precise definition.) The type (15) will, then, give us a type corresponding to the complete Figure 1.2 since it will be the type consisting of strings of one or more events of the type (14).

(15) (pick_up(a,c)⌢attract_attention(a,b)⌢throw(a,c)⌢run_after(b,c)⌢pick_up(b,c)⌢return(b,c,a))$^+$

We will complicate (15) slightly by substituting record types for the ptypes as in (16). We do this because we will want to allow for things happening simultaneously and record types will give us a straightforward way of allowing this.

(16) ($\left[\text{e:pick\_up}(a,c)\right]$⌢$\left[\text{e:attract\_attention}(a,b)\right]$⌢$\left[\text{e:throw}(a,c)\right]$⌢$\left[\text{e:run\_after}(b,c)\right]$⌢$\left[\text{e:pick\_up}(b,c)\right]$⌢$\left[\text{e:return}(b,c,a)\right]$)$^+$

The label ‘e’ (“event”) occurs in each of the elements of the string type. In this case we will say that ‘e’ labels a *dimension* of events of this type. The ‘e’-dimension can be thought of as the dimension which characterizes what is happening at each stage of the event. If you want to think geometrically, you can think of the event-string as being located in a space of event types (that is, the ptypes).

What happens when Kim perceives an event as being of this type? She makes a series of observations of events, assigning them to types in the string type. Note that the ptypes in each of the types can be further broken down in a similar way. This gives us a whole hierarchy of perceived events which at some point have to bottom out in basic perceptions which are not further analyzed. In order to recognize an event as being of this type Kim does not need to perceive a string of events corresponding to each of the types in the string types. She may, for example, observe the boy waving the stick to attract the dog’s attention, get distracted by a bird flying overhead for a while, and then return to the fetch event at the point where the dog is running back to the boy with the stick. This still enables her to perceive the event as an event of fetch playing because she has seen such events before and learned that such events are of the string type in (16). It suffices for her to observe enough of the elements in the string to distinguish the event from other event

[2]This notation was introduced by the mathematician Stephen Kleene.

types she may have available in her knowledge resources. Suppose, for example, that she has just two event string types available that begin with the picking up of a stick by a human in the company of a dog. One is (16). The other is one that leads to the human beating the dog with the stick. If she only observes the picking up of the stick she cannot be sure whether what she is observing is a game of fetch or a beating. However, as soon as she observes something in the event string which belongs only to the fetch type of string she can reasonably conclude that she is observing an event of the fetch type. She may, of course, be wrong. She may be observing an event of a type which she does not yet have available in her resource of event types, in which case she will need to learn about the new event type and add it to her resources. However, given the resources at her disposal she can make a prediction about the nature of the rest of the event. One could model her prediction making ability in terms of a function which maps a situation (modelled as a record) to a type of predicted situation, for example (17).

$$(17)\quad \lambda r{:}\begin{bmatrix} \text{x:}\textit{Ind} \\ \text{c}_{\text{human}}\text{:human(x)} \\ \text{y:}\textit{Ind} \\ \text{c}_{\text{dog}}\text{:dog(y)} \\ \text{z:}\textit{Ind} \\ \text{c}_{\text{stick}}\text{:stick(z)} \\ \text{e:}\left[\text{e:pick\_up(x,z)}\right]^\frown\left[\text{e:attract\_attention(x,y,)}\right] \end{bmatrix}.$$

$$\begin{bmatrix} \text{e:play\_fetch}(r.\text{x},r.\text{y},r.\text{z}) \\ \text{c}_{\text{init}}\text{:init}(r.\text{e,e}) \end{bmatrix}$$

Here the predicate 'init' has arity [*String*, *String*]. The type init(s_1,s_2) is non-empty just in case s_1 is an initial substring of s_2. We achieve this by defining

> If s_1 is a string of length n and s_2 is a string of any length, then s : init(s_1,s_2) iff the length of s_2 is greater than or equal to n and for each i, $0 \leq i < n$, $s_1[i] = s_2[i]$ and $s = s_2$.

That is, if the initial substring condition holds then the second argument to the predicate (and nothing else) is of the ptype.

The kind of function of which (17) is an instance is a function of the general form (18).

$$(18)\quad \lambda a{:}T_1\ .\ T_2(a)$$

where we use the notation $T_2(a)$ to represent the fact that T_2 depends on a. The nature of this dependence in (17) is seen in the occurrences of r in the body of the function, for example,

'play_fetch(r.x,r.y,r.z)'. Such a function maps an object of some type (represented by T_1) to a type (represented by $T_2(a)$). The type that results from an application of this function will depend on what object it is applied to – that is, we have the possibility of obtaining different types from different objects. In type theory such a function is often called a *dependent type*. These functions will play an important role in much of what is to come later in this book. They will show up many times in what appear at first blush to be totally unrelated phenomena. We want to suggest, however, that all of the phenomena we will describe using such functions have their origin in our basic cognitive ability to make predictions on the basis of partial observation of objects and events.

What happens when Kim does not observe enough of the event to be able to predict with any certainty that the complete event will be a game of fetch? One theory would be that she can only make categorical judgements, and that she has to wait until she has seen enough so that there is only one type that matches in the collection of situation types in her resources. Another theory would be one where she predicts a disjunction of the available matching types when there is more than one that matches. One might refine this theory so that she can choose one of the available types but assign it a probability based on the number of matching types. If n is the number of matching types the probability of any one of them might be $\frac{1}{n}$. This assumes that each of the types is equally likely to be realized. It would be natural to assume, however, that the probability which Kim assigns to any one of the matching types would be dependent on her previous experience. Suppose, for example, that she has seen 100 events of a boy picking up a stick in the company of a dog, 99 of those events led to a game of fetch and only one led to the boy beating the dog. One might then assume that when she now sees the boy pick up the stick she would assign a 99% probability to the type of fetch events and only 1% probability to the boy beating the dog. That is, the probability she assigns to an event of a boy picking up a stick leading to a game of fetch is the result of dividing the number of instances of a game of fetch she has already observed by the sum of the number of instances she has observed of any types whose initial segment involves the picking up of a stick. In more general terms we can compute the probability which an agent A assigns on the basis of a string, ω of previous observations to a predicted type T_{pr} given an observed type T_{obs}, $P_{A,\omega}(T_{pr} \mid T_{obs})$, in the case where T_{pr} is a member of the set of alternatives which can be predicted from T_{obs} according to A's resources based on ω, $\mathrm{alt}_{A,\omega}(T_{obs})$, by the following formula:

$$P_{A,\omega}(T_{pr} \mid T_{obs}) = \frac{\mid \{T_{pr}\}^{A,\omega} \mid}{\sum_{T_{alt} \in \mathrm{alt}_{A,\omega}(T_{obs})} \mid \{T_{alt}\}^{A,\omega} \mid}$$

where $\{T\}^{A,\omega}$ is the set of objects of type T observed by A in ω. If T_{pr} is not a member of $\mathrm{alt}_{A,\omega}(T_{obs})$, that is not one of the alternatives, we say that $P_{A,\omega}(T_{pr} \mid T_{obs}) = 0$.

While this is still a rather naive and simple view of how probabilities might be assigned it is not without interest, as shown by the following points:

Probability distributions It will always provide a probability distribution over sets of alternatives, that is,

$$\sum_{T_{pr} \in \mathrm{alt}_{A,\omega}(T_{obs})} P_{A,\omega}(T_{pr} \mid T_{obs}) = 1$$

Alternatives We have assumed a notion of alternatives based on types of completed events for which the observed event is an initial segment but other notions of alternativeness could be considered and perhaps even combined.

Relativity of probability assignments The notion of probability is both agent and resource relative. It represents the probability which an agent will assign to a type when observing a given situation after a previous string of observations. Two agents may assign different probabilities depending on the resources they have available.

Learning Relevant observations will update the probability distributions an agent will assign to a given set of alternatives since the probability is computed on the basis of previous observations of the alternative types.

Kim is not alone in being able to draw conclusions based on partial observations of an event. The dog can do it too. As soon as the boy has raised the stick and attracted the dog's attention the dog is excitedly snapping at the stick and starting to run in the direction in which the boy seems to be about to throw. The dog also seems to be attuned to string types of events just as Kim is and also able to make predictions on the basis of partial observations. The types to which a dog is attuned will not be the same as those to which humans can be attuned and this can certainly lead to miscommunication between humans and dogs. For example, there may be many reasons why I would go to the place where outdoor clothes are hanging and where the dog's lead is kept. Many times it will be because I am planning to take the dog out for a walk, but not as often as the dog appears to think, judging from the excitement he shows any time I go near the lead. It is difficult to explain to the dog that I am just looking for a receipt that I think I might have left in my coat pocket. But the basic mechanism of being able to assemble types of events into string types of more complex events and make predictions on the basis of these types seems to be common to both humans and dogs and a good number of other animals too. Perhaps simple organisms do not have this ability and can only react to events that have already happened, but not to predicted outcomes.

This basic inferential ability is thus not parasitic on the ability to communicate using a human language. It is, however, an ability which appears to be exploited to a great extent in our use of language as we will see in later chapters. In the remaining sections of this chapter we will look

at some aspects of the type theory which seem more likely to correspond to cognitive abilities which only humans have.

1.5 Doing things with types

The boy and the dog have to coordinate and interact in order create an event of the game of fetch. This involves doing more with types than just making judgements. For example, when the dog observes the situation in which the boy raises the stick, it may not be clear to the dog whether this is part of a fetch-game situation or a stick-beating situation. The dog may be in a situation of entertaining these two types as possibilities prior to making the judgement that the situation is of the fetch type. We will call this act a query as opposed to a judgement. Once the dog has made the judgement that what it has observed so far is an initial segment of a fetch type situation it has to make its own contribution in order to realize the fetch type, that is, it has to run after the stick and bring it back. This involves the creation of a situation of a certain type. Thus creation acts are another kind of act related to types. Creating objects of a given type often has a *de se* (see, for example, Perry, 1977; Lewis, 1979; Ninan, 2010; Schlenker, 2011) aspect. The dog has to know that it itself must run after the stick in order to make this a situation in which it and the boy are playing fetch. There is something akin to what Perry calls an essential indexical here, though, of course, the dog does not have indexical linguistic expressions. It is nevertheless part of the basic competence that an agent needs in order to be able to coordinate its action with the rest of the world that it has a primitive sense of self which is distinct from being able to identify an object which has the same properties as itself. We will follow Lewis in modelling *de se* in terms of functional abstraction over the "self". In our terms this will mean that *de se* type acts involve dependent types.

In standard type theory we have judgements such as $o : T$ "o is of type T" and T *true* "there is something of type T". We want to enhance this notion of judgement by including a reference to the agent A which makes the judgement, giving judgements such as $o :_A T$ "agent A judges that o is of type T" and $:_A T$ "agent A judges that there is some object of type T". We will call the first of these a *specific* judgement and the second a *non-specific* judgement. Such judgements are one of the three kinds of acts represented in (19) that we want to include in our type act theory.

(19) *Type Acts*

judgements

specific $o :_A T$ "agent A judges object o to be of type T"

non-specific $:_A T$ "agent A judges that there is some object of type T"

queries

specific $o :_A T?$ "agent A wonders whether object o is of type T"

non-specific $:_A T?$ "agent A wonders whether there is some object of type T"

creations

non-specific $:_A T!$ "agent A creates something of type T"

Note that creations only come in the non-specific variant. You cannot create an object which already exists.

Creations are also limited in that there are certain types which a given agent is not able to realize as the main actor. Consider for example the event type involved in the fetch game of the dog running after the stick. The human cannot be the main creator of such an event since it is the dog who is the actor. The most the human can do is wait until the dog has carried out the action and we will count this as a creation type act. This will become important when we discuss coordination in the fetch-game below. It is actually important that the human makes this passive contribution to the creation of the event of the dog running after the stick and does not, for example, get the game confused by immediately throwing another stick before the dog has had a chance to retrieve the first stick. There are other cases of event types which require a less passive contribution from an agent other than the main actor. Consider the type of event where the dog returns the stick to the human. The dog is clearly the main actor here but the human has also a role to play in making the event realized. For example, if the human turns her back on the dog and ignores what is happening or runs away the event type will not be realized despite the dog's best efforts. Other event types, such as lifting a piano, involve more equal collaboration between two or more agents, where it is not intuitively clear that any one of the agents is the main actor. So when we say "agent A creates something of type T" perhaps it would be more accurate to phrase this as "agent A contributes to the creation of something of type T" where A's contribution might be as little as not realizing any of the other types involved in the game until T has been realized.

De se type acts involve functions which have the agent in its domain and return a type, that is, they are dependent types which, given the agent, will yield a type. We will say that agents are of type *Ind* and that the relevant dependent types, $\mathcal{T}$, are functions of type (*Ind*→*Type*). We characterize *de se* type acts in a way parallel to (19), as given in (20).

(20) *De Se Type Acts*

judgements

specific $o :_A \mathcal{T}(A)$ "agent A judges object o to be of type $\mathcal{T}(A)$"

non-specific $:_A \mathcal{T}(A)$ "agent A judges that there is some object of type $\mathcal{T}(A)$"

queries

specific $o :_A \mathcal{T}(A)?$ "agent A wonders whether object o is of type $\mathcal{T}(A)$"

non-specific $:_A \mathcal{T}(A)?$ "agent A wonders whether there is some object of type $\mathcal{T}(A)$"

creations

non-specific $:_A \mathcal{T}(A)!$ "agent A creates something of type $\mathcal{T}(A)$"

From the point of view of the type theory *de se* type acts seem more complex than non-*de se* type acts since they involve a dependent rather than a non-dependent type and a functional application of that dependent type to the agent. However, from a cognitive perspective one might expect *de se* type acts to be more basic. Agents which perform type acts using types directly related to themselves are behaving egocentrically and one could regard it as a more advanced level of abstraction to consider types which are independent of the agent. This seems a puzzling way in which our notions of type seem in conflict with out intuitions about cognition.

While these type acts are prelinguistic (we need them to account for the dog's behaviour in the game of fetch) we will try to argue later that they are the basis on which the notion of speech act (Austin, 1962; Searle, 1969) is built. Our notion of using types in query acts seems intuitively related to work on inquisitive semantics (Groenendijk and Roelofsen, 2012) where some propositions (in particular disjunctions) are regarded as inquisitive. However, this will still allow us to make a distinction between questions and assertions in natural language as argued for by Ginzburg (2012).

Let us now apply these notions to the kind of interaction that has to take place between the human

and the dog in a game of fetch. First consider in more detail what is actually involved in playing a game of fetch, that is creating an event of type (16). Each agent has to keep track in some way of where they are in the game and in particular what needs to happen next. We analyze this by saying that each agent has an information state which we will model as a record. We need to keep track of the progression of types of information state for an agent during the course of the game. We will refer to the types of information states as gameboards.[3] The idea is that as part of the event occurs then the agent's gameboard is updated so that an event of the next type in the string is expected. For now, we will consider gameboards which only place one requirement on information states, namely that there is an agenda which indicates the type of the next move in the game. Thus if the agent is playing fetch and observes an event of the type where the human throws the stick, then, according to (16), the next move in the game will be an event of the type where the dog runs after the stick. If the actor in the next move is the agent herself then the agent will need to create an event of the type of the next move if the game is to progress. If the actor in the next move is the other player in the game, then the agent will need to observe an event and judge it to be of the appropriate type in order for the game to progress. The type of information states, *InfoState*, will be (21a). (In Chapter 2, when we apply these ideas to dialogue, we will see more complex information states.) The type of the initial information state, *InitInfoState*, will be one where the agenda is required to be the empty list.

(21) a. $\left[\text{agenda} : [\mathit{RecType}] \right]$

b. $\left[\text{agenda=[]} : [\mathit{RecType}] \right]$

We can now see the rules of the game corresponding to the type (16) as a set of update functions which indicate for an information state of a given type what type the next information state may belong to if an event of a certain type occurs. These update functions correspond to the transitions in a finite state machine. This is given in (22).

[3]Our notions of *information state* and *gameboard* are taken from Larsson (2002) and Ginzburg (2012) respectively as well as a great deal of related literature on the gameboard or information state approach to dialogue analysis originating from Ginzburg (1994). We have adapted the notions somewhat to our own purposes and will take this up in more detail in Chapter 2.

(22) { λr:[agenda=[]:[*RecType*]] .
[agenda=[[e:pick_up(a,c)]]:[*RecType*]],
λr:[agenda=[[e:pick_up(a,c)]]:[*RecType*]]
λe:[e:pick_up(a,c)] .
[agenda=[[e:attract_attention(a,b)]]:[*RecType*]],
λr:[agenda=[[e:pick_up(a,c)]]:[*RecType*]]
λe:[e:attract_attention(a,b)] .
[agenda=[[e:throw(a,c)]]:[*RecType*]],
λr:[agenda=[[e:throw(a,c)]]:[*RecType*]]
λe:[e:throw(a,c)] .
[agenda=[[e:run_after(b,c)]]:[*RecType*]],
λr:[agenda=[[e:run_after(b,c)]]:[*RecType*]]
λe:[e:run_after(b,c)] .
[agenda=[[e:pick_up(b,c)]]:[*RecType*]],
λr:[agenda=[[e:pick_up(b,c)]]:[*RecType*]]
λe:[e:pick_up(b,c)] .
[agenda=[[e:return(b,c,a)]]:[*RecType*]],
λr:[agenda=[[e:return(b,c,a)]]:[*RecType*]]
λe:[e:return(b,c,a)] .
[agenda=[]:[*RecType*]]
}

Since we are treating an empty agenda as the condition for the input to the initial state in the corresponding automaton and also the output of the final state we automatically get the loop effect from the final state to the initial state. In order to prevent the loop we would have to distinguish the type corresponding to the initial and final states. Note that the functions in (22) are of the type (23).

(23) ([agenda:[*RecType*]]$\rightarrow$(*Rec*$\rightarrow$*RecType*))

That is, they map an information state containing an agenda (modelled as a record containing an agenda field) and an event (modelled as a record) to a record type. This is true of all except for the function corresponding to the initial state which is of type (24).

(24) ([agenda:[*RecType*]]$\rightarrow$*RecType*)

That is, it maps an information state directly to a record type and does not require an event. We can think of this set as the set of rules which define the game. It is of the type (25).

(25) $\{((\left[\text{agenda:[}\mathit{RecType}\text{]}\right]\rightarrow(\mathit{Rec}\rightarrow\mathit{RecType}))\vee(\left[\text{agenda:[}\mathit{RecType}\text{]}\right]\rightarrow\mathit{RecType}))\}$

Let us call the type in (25) *GameRules*. Sets of game rules of this type define the rules for specific participants as in (22). In order to characterize the game in general we need to abstract out the roles of the individual participants in the game. This we will do by defining a function from a record containing individuals appropriate to play the roles in the game thus revising (22) to (26).

(26) $\lambda r^*\text{:}\begin{bmatrix} \text{h} & : & \mathit{Ind} \\ \text{c}_{\text{human}} & : & \text{human(h)} \\ \text{d} & : & \mathit{Ind} \\ \text{c}_{\text{dog}} & : & \text{dog(d)} \\ \text{s} & : & \mathit{Ind} \\ \text{c}_{\text{stick}} & : & \text{stick(s)} \end{bmatrix}$.

$\{\ \lambda r\text{:}\left[\text{agenda=[]:[}\mathit{RecType}\text{]}\right]$.
$\left[\text{agenda=[}\left[\text{e:pick\_up(}r^*\text{.h,}r^*\text{.s)}\right]\text{]:[}\mathit{RecType}\text{]}\right]$,
$\lambda r\text{:}\left[\text{agenda=[}\left[\text{e:pick\_up(}r^*\text{.h,}r^*\text{.s)}\right]\text{]:[}\mathit{RecType}\text{]}\right]$
$\lambda e\text{:}\left[\text{e:pick\_up(}r^*\text{.h,}r^*\text{.s)}\right]$.
$\left[\text{agenda=[}\left[\text{e:attract\_attention(}r^*\text{.h,}r^*\text{.d)}\right]\text{]:[}\mathit{RecType}\text{]}\right]$,
$\lambda r\text{:}\left[\text{agenda=[}\left[\text{e:pick\_up(}r^*\text{.h,}r^*\text{.s)}\right]\text{]:[}\mathit{RecType}\text{]}\right]$
$\lambda e\text{:}\left[\text{e:attract\_attention(}r^*\text{.h,}r^*\text{.d)}\right]$.
$\left[\text{agenda=[}\left[\text{e:throw(}r^*\text{.h,}r^*\text{.s)}\right]\text{]:[}\mathit{RecType}\text{]}\right]$,
$\lambda r\text{:}\left[\text{agenda=[}\left[\text{e:throw(}r^*\text{.h,}r^*\text{.s)}\right]\text{]:[}\mathit{RecType}\text{]}\right]$
$\lambda e\text{:}\left[\text{e:throw(}r^*\text{.h,}r^*\text{.s)}\right]$.
$\left[\text{agenda=[}\left[\text{e:run\_after(}r^*\text{.d,}r^*\text{.s)}\right]\text{]:[}\mathit{RecType}\text{]}\right]$,
$\lambda r\text{:}\left[\text{agenda=[}\left[\text{e:run\_after(}r^*\text{.d,}r^*\text{.s)}\right]\text{]:[}\mathit{RecType}\text{]}\right]$
$\lambda e\text{:}\left[\text{e:run\_after(}r^*\text{.d,}r^*\text{.s)}\right]$.
$\left[\text{agenda=[}\left[\text{e:pick\_up(}r^*\text{.d,}r^*\text{.s)}\right]\text{]:[}\mathit{RecType}\text{]}\right]$,
$\lambda r\text{:}\left[\text{agenda=[}\left[\text{e:pick\_up(}r^*\text{.d,}r^*\text{.s)}\right]\text{]:[}\mathit{RecType}\text{]}\right]$
$\lambda e\text{:}\left[\text{e:pick\_up(}r^*\text{.d,}r^*\text{.s)}\right]$.
$\left[\text{agenda=[}\left[\text{e:return(}r^*\text{.d,}r^*\text{.s,}r^*\text{.h)}\right]\text{]:[}\mathit{RecType}\text{]}\right]$,
$\lambda r\text{:}\left[\text{agenda=[}\left[\text{e:return(}r^*\text{.d,}r^*\text{.s,}r^*\text{.h)}\right]\text{]:[}\mathit{RecType}\text{]}\right]$
$\lambda e\text{:}\left[\text{e:return(}r^*\text{.d,}r^*\text{.s,}r^*\text{.h)}\right]$.
$\left[\text{agenda=[]:[}\mathit{RecType}\text{]}\right]$
$\}$

(26) is of type (*Rec*→*GameRules*) which we will call *Game*.

Specifying the rules of the game in terms of update functions in this way will not actually getting anything to happen, though. For that we need type acts of the kind we discussed. We link the update functions to type acts by means of *licensing conditions on type acts*. A basic licensing

condition is that an agent can create (or contribute to the creation of) a witness for the first type that occurs on the agenda in its information state. Such a licensing condition is expressed in (27).

(27) If A is an agent, s_i is A's current information state, $s_i :_A \left[\text{ agenda=}T \,|\, R \quad : \quad [\textit{RecType}] \right]$, then $:_A \; T!$ is licensed.

Update functions of the kind we have discussed are handled by the licensing conditions in (28).

(28) a. If $f : (T_1 \rightarrow (T_2 \rightarrow \textit{Type}))$ is an update function, A is an agent, s_i is A's current information state, $s_i :_A T_i$, $T_i \sqsubseteq T_1$ (and $s_i : T_1$), then an event $e :_A T_2$ (and $e : T_2$) licenses $s_{i+1} :_A f(s_i)(e)$.

b. If $f : (T_1 \rightarrow (T_2 \rightarrow \textit{Type}))$ is an update function, A is an agent, s_i is A's current information state, $s_i :_A T_i$, $T_i \sqsubseteq T_1$ (and $s_i : T_1$), $s_{i+1} :_A f(s_i)$ is licensed.

(28a) is for the case where the update function requires an event in order to be triggered and (28b) is for the case where no event is required. There are two variants of licensing conditions which can be considered. One variant is where the licensing conditions rely only on the agent's judgement of information states and events occurring. The other variant is where in addition we require that the information states and events actually are of the types which the agent judges them to be of. (These conditions are represented in parentheses in (28).) In practical terms an agent has to rely on its own judgement, of course, and there is one sense in which any resulting action is licensed even if the agent's judgement was mistaken. There is another stricter sense of license which requires the agent's judgement to be correct. In the real world, though, the only way we have of judging a judgement to be correct is to look at judgements by other agents.

Licensing conditions will regulate the coordination of successfully realized games like fetch. They enable the agents to coordinate their activity when they both have access to the same objects of type *Game* and are both willing to play. The use of the word "license" is important, however. The agents have free will and may choose not to do what is licensed and also may perform acts that are not licensed. We cannot build a theory that will predict exactly what will happen but we can have a theory which tells us what kinds of actions belong to a game. It is up to the agents to decide whether they will play the game or not. At the same time, however, we might regard whatever is licensed at a given point in the game as an obligation. That is, if there is a general obligation to continue a game once you have embarked on it, then whatever type is placed on an

agent's agenda as the result of a previous event in the game can be seen as an obligation on the agent to play its part in the creation of an event of that type.

1.6 Modal type systems

Kim continues her walk still thinking about the boy and the dog. She thinks, "Was the boy standing too close to the pond? Suppose he had fallen in. If he had been my son, I wouldn't have let him play just there." An important aspect of human cognition is that we are not only able observe things as they are but also to conceive of alternatives which go beyond the completion of observed events in the way discussed in Section 1.4. We can not only observe objects and perceive them to be of certain types we can also consider possibilities in which they belong to different types and perhaps do not belong to the type we have observed. We have managed to unhook type judgements from direct perception. While the seeds of this ability can be seen in the kind of event perception and prediction discussed above in that it gives us a way to consider types which have not yet been realized, it is at least one step further in cognitive evolution to be able to consider alternative type assignments which do not correspond to completions of events already perceived.

This leads us to construct *modal type systems* with alternative assignments of objects to types.[4] Figure 1.4 provides an example of a modal system of basic types with two possibilities, one where the extensions of types T_1 and T_2 overlap and another possibility where they do not.

The object a is of type T_1 in the first possibility but not in the second possibility. There is an object, b, of type T_1 in the second possibility. b does not exist at all in the first possibility. In the figure we just show two possibilities but our general definition in Appendix A.2 allows for there to be any number of possibilities, including infinitely many.

Given this apparatus we define four simple modal notions:

(necessary) equivalence Two types are (necessarily) equivalent just in case the extension of one types is identical with that of the other type in all the possibilities. While the different possibilities may provide different extensions for the types, it will always be the case that in any given possibility the two types will have the same extension.

subtype One type is a subtype of another just in case whatever possibility you look at it is always the case that the extension of the first type is a subset of the extension of the second. We can also say that the first type "entails" the second, that is, any object which is of the first type will also be of the second type, no matter which possibility you are considering.

[4]The term *modal* is taken from modal logic. See Hughes and Cresswell (1968) for a classic introduction. A modern introduction is to be found in Blackburn *et al.* (2001).

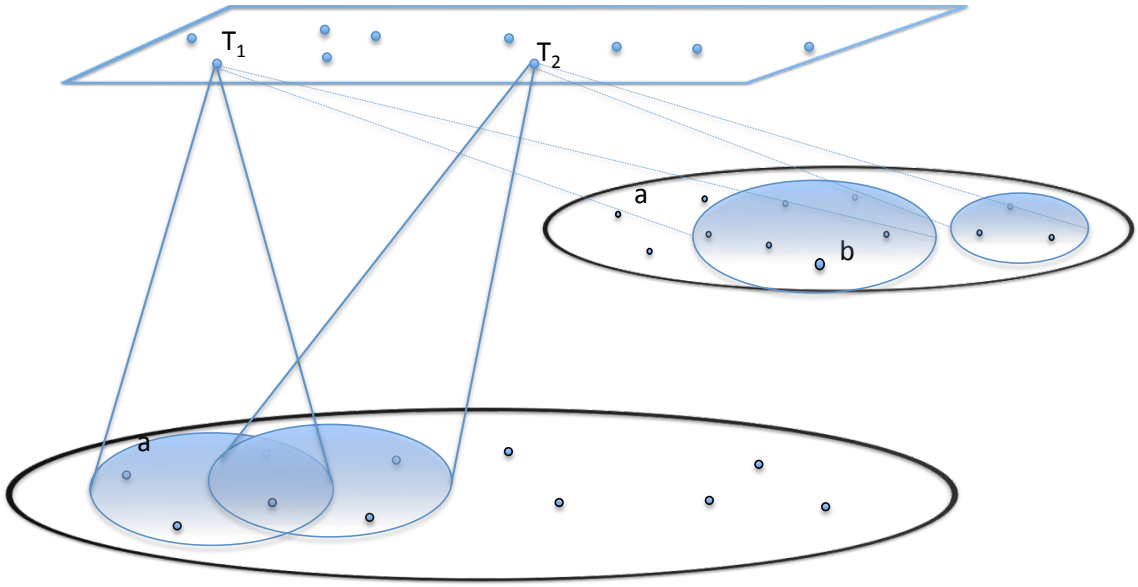


Figure 1.4: Modal system of basic types

necessity The notion of necessity we characterize for a type could be glossed as "necessarily realized" or "necessarily instantiated". A type will be necessary just in case there is something of the type in all the possibilities.

possibility This notion corresponds to "possibly realized" or "possibly instantiated". A type will be possible just in case there is some possibility according to which it has a non-null extension.

These notions are made precise in Appendix A.2. Note that all of these notions are relativized to the modal system you are considering and the possibilities it offers. We may think of the family of assignments $\mathcal{A}$ as providing a modal base (cf. Kratzer) or alternatives (in the sense of ????). For these kinds of applications we may wish to consider very small families of assignments corresponding to the knowledge we have. Alternatively, we may want to consider strong logical variants of these modal notions where we consider all the logical possibilities, for example, all possible assignments of extensions to types.

So far we have talked about modal systems of basic types. Modal systems of complex types, where we introduce ptypes, create a minor complication. What ptypes that are present in a system depends on what objects there are of the types that are used in the arities of the predicates. Thus if we have some predicate r with arity $\langle \mathit{Ind}, \mathit{Ind} \rangle$ and a possibility where the set assigned to *Ind* is $\{a, b\}$ then according to that possibility the ptypes formed with r will be $r(a, a)$, $r(a, b)$, $r(b, a)$ and $r(b, b)$. In a possibility where *Ind* is assigned a different set the set of available ptypes

will be different. It is an important feature of type theories with types constructed from predicates that the collection of such types depends on what objects are available as arguments to the predicates. This makes type theory very different from a logical language such as predicate calculus where the notion of well-formedness of syntactic expressions containing predicates is defined independently of what is provided by the model as denotations of arguments to the predicate.

This leads us (in Appendix A.9) to define two variants of each of our modal notions: *restrictive* variants which are only defined for types which exist in all possibilities and *inclusive* variants which require that the modal definition holds for all the possibilities in which the types exist and disregards those in which the types do not exist. For example, a type is *necessary*$_r$ (that is, "restrictively necessary") just in case the type is available in all possibilities and has a non-empty set of witnesses in all possibilities. It is *necessary*$_i$ ("inclusively necessary") just in case in all the possibilities in which the type is provided it has a non-empty set of witnesses. It is clear that if a type is necessary$_r$ it will also be necessary$_i$ but there may be types which are necessary$_i$ but not necessary$_r$ (if the type is not provided in all possibilities). A similar relationship between the restrictive and inclusive notions holds for all the modal notions we have discussed.

There may be significant classes of modal type systems in which the types available in the different possibilities do not vary. This could be achieved by requiring that the types used in the arities of predicates always have the same witnesses in all the possibilities. This seems feasible if we restrict the types used in predicate arities to basic ontological categories such as individual or time point. It seems reasonable to consider modal systems in which an individual in one possibility will be an individual in any other possibility, for example. It seems reasonable to say that we wish to consider possibilities where, for example, Kim is a man rather than a woman, but not possibilities where Kim is a point in time rather than an individual. However, the notion "basic ontological category" is a slippery one and we do not want to be forced to make commitments about that.

In the definition of a system of complex types in section A.3.2 we call the pair of an assignment to basic types and assignment to ptypes, $\langle A, F\rangle$, a *model* because of its similarity to first order models.[5] The model provides an interface between the type theoretical system and a domain external to the type theory. The natural domain to relate to the type theory is that of individuals and situations, that is the kind of things we can perceive or at least consider as possibilities. However, we may want to use models which relate to our perceptual apparatus, as in Larsson (2011), rather than directly to the world. This can also be the key for relating the type theory to a dynamically changing world where the models representing our perceived possibilities are not fixed. We will return to this in Chapter 5.

[5]For a more detailed discussion of the relationship between this and first order models as used in the interpretation of first order logic see Cooper (fthc).

1.7 Intensionality: propositions as types

Kim continues to think about the boy and the dog as she walks along. It was fun to see them playing together. They seemed so happy. The boy obviously thought that the dog was a good playmate. Kim is not only able to perceive events as being of certain types. She is able to recall and reflect on these types. She is able to form attitudes towards these types: it was fun that the boy and the dog were playing but a little worrying that they were so close to the pond. This means that the types themselves seem to be arguments to predicates like 'fun' and 'worrying'. This seems to be an important human ability – not only to be able to take part in or observe an event and find it fun or worrying but to be able to reflect independently of the actual occurrence of the event that it or in general similar events are fun or worrying. This is a source of great richness in human cognition in that it enables us to consider situation types independently of their actual instantiation.[6] This abstraction also enables us to consider what attitudes other individuals might have. For example, Kim believes that the boy thought that the dog was a good playmate. She is able to ascribe this belief to the boy. Furthermore, we are able to reflect on Kim's state of mind where she has a belief concerning the type of situation where the boy thinks that the dog was a good playmate. And somebody else could consider of us that we have a certain belief about Kim concerning her belief about the boy's belief. There is in principle no limit to the depth of recursion concerning our attitudes towards types.

We propose to capture this reflective nature of human cognition by making the type theory technically *reflective* in the sense that we allow types themselves to be objects which can belong to other types. In classical model theoretic semantics we think of *believe* as corresponding to a relation between individuals and propositions. In our type theory, however, we are subscribing to the "propositions as types" view which comes to us via Martin-Löf (1984) but has its origins in intuitionistic logic [????]. Propositions are true or false. Types of situations such as hug(a,b) correspond to propositions in the sense that is they are non-empty then the proposition is true. If there is nothing of this type then it is false. The reasoning is thus that we do not need propositions in our system as separate semantic objects if we already have types. We can use the types to play the role of propositions. To believe a type is to believe it to be non-empty. From the point of view of a type theory for cognition in which we connect types to our basic perceptual ability, this provides a welcome link between our perceptual ability and our ability to entertain propositions (that is, to consider whether they are true or false).

A predicate like 'believe' which represents that an individual has an attitude (of belief) to a certain type should thus have an arity which requires its arguments to be an individual and a type. That is, we should be able to construct the type believe(c, hug(a,b)) corresponding to c *believes that* a *hugs* b. We thus create *intensional* type systems where types themselves can be treated as objects and belong to types. Care has to be taken in constructing such systems in order

[6] This richness also has its downside in that we often become so engaged in our internal cognitive abstraction that it can be difficult to be fully present and conscious of our direct perception of the world – for example, worrying about what might happen in the future rather than enjoying the present.

to avoid paradoxes. We use a standard technique known as stratification Turner (2005). We start with a basic type system and then add higher order levels of types. Each higher order includes the types of the order immediately below as objects. In each of these higher orders n there will be a type of all types of the order $n-1$ but there is no ultimate "type of all types" – such a type would have to have itself as an object. This is made precise in Appendix A.10. For more detailed discussion see Cooper (fthc). Figure 1.5 represents an intensional modal type system where we indicate just the initial three orders of an infinite hierarchy of type orders.

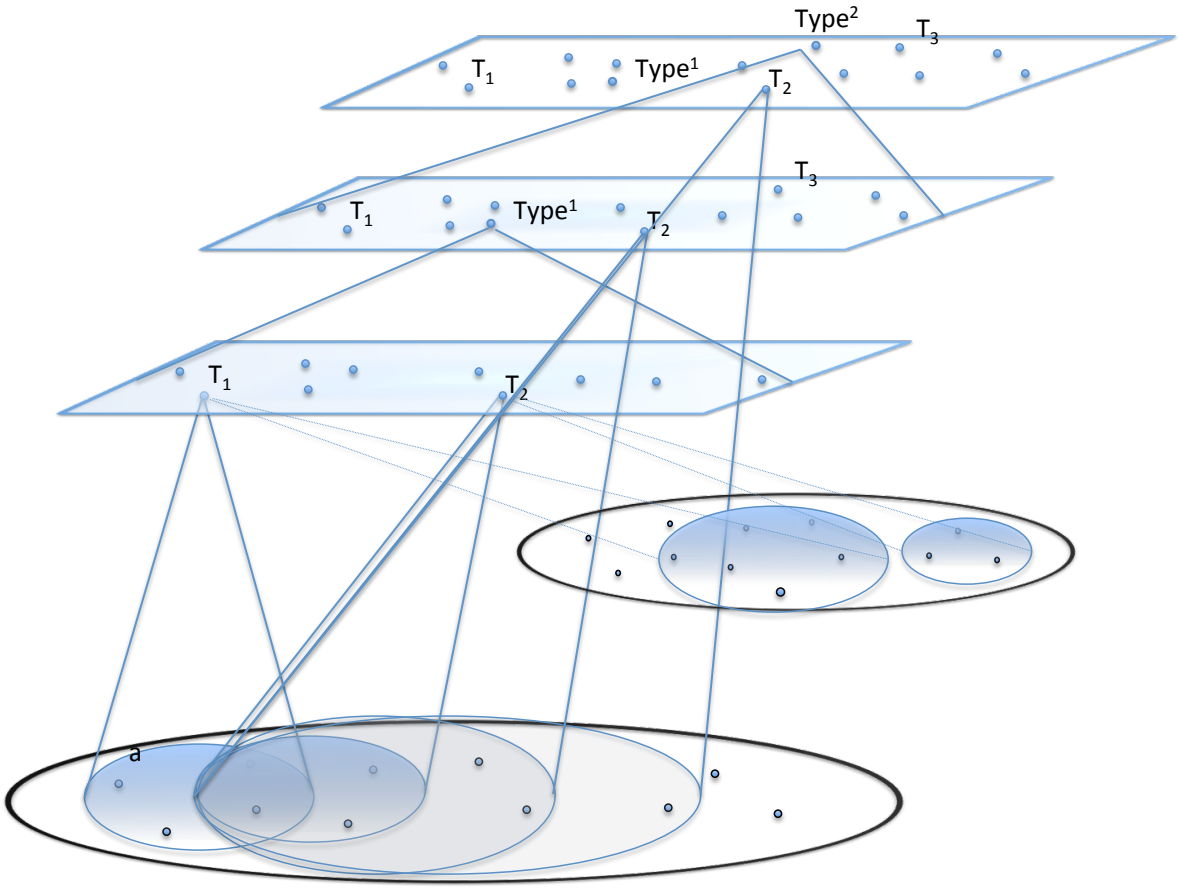


Figure 1.5: Intensional modal type system

Chapter 2

Information exchange

2.1 Speech events

In Chapter 1 we talked about the perception of events such as a boy and a dog playing fetch. We imagined Kim walking through the park and perceiving various kinds of events. Suppose that she meets a friend in the park and they start to have a conversation. A conversation is a kind of event involving language which seems to be uniquely human. The kind of dialogue involved in a conversation enables humans to exchange information in a way that is more complex and more abstracted from currently occurring events than other animals seem capable of. Nevertheless, we will argue that the basic mechanisms of dialogue involve assigning types to events in way that we discussed in Chapter 1. The events involved are *speech events*.

Consider the kind of event type prediction that we considered in Chapter 1. Suppose that Kim sees the boy playing fetch with the dog and the boy is standing close to the lake with his back to it. As the dog runs towards him with the stick he takes a step backwards. "No," says Kim, seeing that the boy is about to fall in the lake. "Watch out," she shouts to the boy who takes a step forward just in time and narrowly misses falling in the lake. Her utterance of *no* represents a negative attitude towards a predicted outcome. This kind of negation is discussed briefly in Cooper and Ginzburg (2011a,b) where examples are given of cases where *no* is a response to a completed event and where it is used as an attempt to prevent the predicted outcome. This latter exploits the fact that agents cannot only perceive and classify events according to the types to which they are attuned but can also intervene and prevent a predicted outcome. Kim's linguistic utterance of *watch out* is used in this way. While Kim is using words of English this is not yet completely linguistic interaction. A dog, sensing danger, will begin to bark and this can have the effect of preventing a predicted outcome. It is a kind of inter-agent communication nevertheless in that it is an intervention in the flow of events which involves predicting and changing the behaviour of

(1)

John:	Hello doctor.
Anon 1:	Hello.
	Well Mr [last or full name], what can I [do for you today]$_1$?
John:	[Er, it's]$_1$ a wee problem I've had for a ⟨pause⟩ say about a year now.
Anon 1:	Mhm.
John:	It's er my face.
	And my skin.
	I seem to get an awful lot of, it's like
Anon 1:	Aha.
John:	dry flaky skin.
Anon 1:	Yeah.
John:	And I get it on my forehead, [down here]$_2$
Anon 1:	[I can see]$_2$

BNC file G43, sentences 1–13

another agent. In this sense it is similar to human dialogue, although human dialogue is normally a much more abstract affair, involving predicting and influencing the other agent's linguistic behaviour and the attitudes and beliefs which the other agent has concerning certain types.

Dialogues themselves are events and, just like other events, can be regarded as strings of smaller events. Consider the dialogue excerpt (1) from the British National Corpus which is the beginning of a consultation between a patient (John) and a doctor (Anon 1).

We might assign the whole dialogue of which this is a part to a genre type for patient doctor consultation.[1] The genre type could be seen as an event type which, like the type for the game of fetch discussed in Chapter 1, can be broken down into a string of subevent types such as greeting (realized here by the exchange *Hello doctor./Hello*), establishing the patient's symptoms (realized here by the remainder of (1)), making a diagnosis, prescribing treatment and so on. Events belonging to these subtypes can be further broken down into strings of turns which further can be broken down into strings of utterances of phrases. In turn phrase utterances are constituted by strings of word utterances which in turn can be regarded as strings of phoneme events. Notice that the temporal relationships between the elements of these strings is more varied than we accounted for in Chapter 1. In dialogue utterances may temporally overlap each other (as indicated in (1) by the notation $[\dots]_n$). When we consider adjacent phoneme events in a string overlap becomes the norm (referred to as coarticulation in phonetics). Although we did not take it up in Chapter 1, temporal overlap in event strings is not restricted to speech events. For example, in the game of fetch it is quite often the case that the dog will start running after the stick before the human has finished throwing it. Perceiving temporally overlapping events is part of our basic perceptual

[1] For a recent discussion of genre in the kind of framework that we are describing see Ginzburg (2010, 2012).

apparatus.

We will work on developing a type for speech events, *SEvent*.[2] Crucial here is the type of phonological event, *Phon*, that is the type of event where certain speech sounds are produced. A field for events of this type will play a role corresponding to the phonology feature in HPSG (Sag *et al.*, 2003). For simplicity we might assume that *Phon* is an abbreviation for $\left[\text{e:}\mathit{Word}\right]^{+}$ that is a non-empty string of events where a word is uttered.[3] Here *Word* is the type of event where word forms of the language are uttered. A more accurate proposal might be that *Phon* is $\left[\text{e:}\mathit{Phoneme}\right]^{+}$[4] where *Phoneme* is the type of utterance event where a phoneme is uttered. This would still be a simplification and an abstraction from the actual events that are being classified, however. A phoneme type is rather to be regarded as a complex type of acoustic and articulatory event and what we regard as a string of phonemes is in fact a string of events where the phoneme types overlap (corresponding to what is know as *coarticulation* in phonological and phonetic theory). For example, the pronunciation of the phoneme /k/ in "kit" is distinct from its pronunciation in "cat" due to the influence of the following vowel. Suppose that the dimensions of phoneme utterance events are given by place, manner, rounding, voicing and nasal. Then we might represent the type of an utterance of /k/ as

(2)
$$\begin{bmatrix} \text{place} & : & \mathit{Velar} \\ \text{manner} & : & \mathit{Stop} \\ \text{rounding} & : & \mathit{NonRound} \\ \text{voicing} & : & \mathit{NonVoiced} \\ \text{nasal} & : & \mathit{NonNasal} \end{bmatrix}$$

the type of an utterance of /i/ by

(3)
$$\begin{bmatrix} \text{place} & : & \mathit{FrontHigh} \\ \text{manner} & : & \mathit{Vocalic} \\ \text{rounding} & : & \mathit{NonRound} \\ \text{voicing} & : & \mathit{Voiced} \\ \text{nasal} & : & \mathit{NonNasal} \end{bmatrix}$$

and the type of an utterance of /æ/ by

[2]This type will be different for different languages, dialects, even idiolects. Thus there will be a different type corresponding to what we think of as speech events of English as opposed to speech events of French. Similar remarks can be made about all the linguistic types that we introduce. We will ignore this in our grammatical types in order to avoid proliferation of subscripts.

[3]If we want to be more grammatically sophisticated we might want to allow silent speech events by allowing empty phonologies, that is, we say that *Phon* is the type $\left[\text{e:}\mathit{Word}\right]^{*}$.

[4]Or $\left[\text{e:}\mathit{Phoneme}\right]^{*}$.

(4)
$$\begin{bmatrix} \text{place} & : & \textit{BackHigh} \\ \text{manner} & : & \textit{Vocalic} \\ \text{rounding} & : & \textit{NonRound} \\ \text{voicing} & : & \textit{Voiced} \\ \text{nasal} & : & \textit{NonNasal} \end{bmatrix}$$

Naively, one might think that the type of the phoneme string /ki/ would be

(5)
$$\begin{bmatrix} & & \begin{bmatrix} \text{place} & : & \textit{Velar} \\ \text{manner} & : & \textit{Stop} \\ \text{rounding} & : & \textit{NonRound} \\ \text{voicing} & : & \textit{NonVoiced} \\ \text{nasal} & : & \textit{NonNasal} \end{bmatrix} \\ \text{e} & : & \end{bmatrix} \frown \begin{bmatrix} & & \begin{bmatrix} \text{place} & : & \textit{FrontHigh} \\ \text{manner} & : & \textit{Vocalic} \\ \text{rounding} & : & \textit{NonRound} \\ \text{voicing} & : & \textit{Voiced} \\ \text{nasal} & : & \textit{NonNasal} \end{bmatrix} \\ \text{e} & : & \end{bmatrix}$$

However, the place of articulation of the /k/ will be influenced by the place of articulation of the following vowel as in (6)

(6)
$$\begin{bmatrix} & & \begin{bmatrix} \text{place} & : & \textit{Palatal} \\ \text{manner} & : & \textit{Stop} \\ \text{rounding} & : & \textit{NonRound} \\ \text{voicing} & : & \textit{NonVoiced} \\ \text{nasal} & : & \textit{NonNasal} \end{bmatrix} \\ \text{e} & : & \end{bmatrix} \frown \begin{bmatrix} & & \begin{bmatrix} \text{place} & : & \textit{FrontHigh} \\ \text{manner} & : & \textit{Vocalic} \\ \text{rounding} & : & \textit{NonRound} \\ \text{voicing} & : & \textit{Voiced} \\ \text{nasal} & : & \textit{NonNasal} \end{bmatrix} \\ \text{e} & : & \end{bmatrix}$$

In addition to this the voice onset associated with the vowel will normally begin before the articulation of the stop is complete as in (7).

(7)
$$\begin{bmatrix} & & \begin{bmatrix} \text{place} & : & \textit{Palatal} \\ \text{manner} & : & \textit{Stop} \\ \text{rounding} & : & \textit{NonRound} \\ \text{voicing} & : & \textit{NonVoiced} \frown \textit{Voiced} \\ \text{nasal} & : & \textit{NonNasal} \end{bmatrix} \\ \text{e} & : & \end{bmatrix} \frown \begin{bmatrix} & & \begin{bmatrix} \text{place} & : & \textit{FrontHigh} \\ \text{manner} & : & \textit{Vocalic} \\ \text{rounding} & : & \textit{NonRound} \\ \text{voicing} & : & \textit{Voiced} \\ \text{nasal} & : & \textit{NonNasal} \end{bmatrix} \\ \text{e} & : & \end{bmatrix}$$

This is not meant to be a serious phonological analysis. We include it here to show how the well-studied phenomenon of coarticulation could be included in the general framework and to show that the notion of overlapping events which we will need later for semantics and dialogue is the same notion that is needed for phonology. We have no more to say about phonology and will limit our analysis of phonological events to strings of words.

We will keep the simplifying assumption that phonology is a string of words here (that is, that *Phon* is *Word*$^+$ and we do not say more about what is of type *Word*) as we do not aim to give a detail account of phonology. Thus a proposal for the type *SEvent* might be (8).

(8) $\left[\begin{array}{lcl} \text{e} & : & \textit{Phon} \end{array}\right]$

To this we might usefully add the speech location as in (9).

(9) $\left[\begin{array}{lcl} \text{e-loc} & : & \textit{Loc} \\ \text{e} & : & \textit{Phon} \\ \text{c}_{\text{loc}} & : & \text{loc(e,e-loc)} \end{array}\right]$

We will take *Loc* to be the type of regions in three dimensional space without specifying more detail. Further if e is an event and l a location we will say that the type loc(e,l) is non-empty just in case e is located at l, again without saying exactly what that means for now.

It might seem natural to add roles of speaker and audience, given what we know about speech act theory (Searle, 1969). Thus we might consider *SEvent* to be the type in (10).

(10) $\left[\begin{array}{lcl} \text{e-loc} & : & \textit{Loc} \\ \text{sp} & : & \textit{Ind} \\ \text{au} & : & \textit{Ind} \\ \text{e} & : & \textit{Phon} \\ \text{c}_{\text{loc}} & : & \text{loc(e,e-loc)} \\ \text{c}_{\text{sp}} & : & \text{speaker(e,sp)} \\ \text{c}_{\text{au}} & : & \text{audience(e,au)} \end{array}\right]$

However, while many speech events may be considered to be of this type, not all will. Of course, some speech events are not addressed to any audience. An example might be an exclamation uttered after hitting one's thumb with a hammer. Longer speech events like dialogues will not have a single speaker or audience. Even shorter chunks corresponding perhaps to single speech acts do not always have a single speaker or audience. For example, consider split utterances as discussed by Purver *et al.* (2010) who give the example (11).

(11) A: I heard a shout. Did you
B: Burn myself? No, luckily.

Here we probably want to consider the utterance of *Did you . . . burn myself?* as a speech event on which A and B collaborate. Otherwise it might be hard to explain how *you* can be interpreted as the subject of *burn*. We have a single predication split across two speakers. Similarly, speakers can address different audiences within the same predicate structure as in (12).

(12) You [pointing] work with you [pointing] and you [pointing] work on your own.

Nevertheless, we might consider that the majority of speech events would belong to the more restricted type (10).

Because we have taken a neo-Davidsonian (Dowty, 1989) approach to the more restricted speech-event types, where the objects playing the various roles in the speech events are introduced in separate fields, both (9) and (10) are subtypes of (8). We will use *SEvent* below to represent the most specific of the types, (10), while bearing in mind that many events we may want to call "speech events" will belong only to more general types such as (9) and (8).

2.2 Signs

We interpret many speech events as being associated with a semantic content, but not all. When John in (1) says *It's a wee problem I've had for a, say, about a year now* he is using the speech event to refer to another situation - a situation in which he has dry skin for a period of a year. This is what Barwise and Perry (1983) would refer to as the *described situation* which is distinct from the speech situation. In contrast the doctor's utterance of *Hello* in (1) does not tell us anything about a described situation external to the current conversation, although it does give us information about where we are in the conversation (the beginning) and indicate that the doctor is paying attention. We shall say that the former utterance with a type of described situation and call this the *content* of the utterance. A situation type is an appropriate content for a declarative sentence used to make an assertion.[5] The contents of phrases within such a sentence such as *a wee problem* or *about a year* will be objects which can be combined to produce such a type. The contents of other kinds of speech acts, for example, associated with questions like the doctor's utterance of *what can I do for you today?* will be objects based on situation types, in the case of this question a function which maps actions to a situation type. (See Ginzburg (2012) for a discussion of the kind of treatment of questions we have in mind.)

We can think of this association of content with a speech event in similar terms to prediction of event completion discussed in Section 1.4 of Chapter 1. At least in the case of declarative

[5]We will discuss later that alternative proposed in Ginzburg (2012) that it should be a pairing of a situation type with a situation, that is an Austinian proposition as introduced by Barwise and Perry (1983) based on Austin (1961).

assertions it is a mapping from an observation of a situation to a type of situation. In the case of the event completion the result of the mapping was a type for the completion of the event so far observed. In the case of the speech event we are relating the observation to a type of situation which is entirely distinct from the speech event. The association is less immediate and more abstract but the underlying mechanism, associating the observation of a situation of a given type with another type and drawing the conclusion that the second type must be non-empty, is the same. We could represent the association by a function of the form (13), corresponding to (18) in Chapter 1.

(13) $\lambda s{:}T_{SpEv} \; . \; T_{Cnt}(s)$

This represents a mapping from a speech event s of a given type T_{SpEv} to a type T_{Cnt} which is the content of the speech event. The type T_{Cnt} can depend on s (for example, the type of the described situation may require that the described situation be related to the utterance situation temporally or spatially).

de Saussure (1916) called the association between speech and content a sign and this notion has been taken up in modern linguistics in Head Driven Phrase Structure Grammar (HPSG, Sag *et al.*, 2003). In HPSG a sign is regarded technically as a feature structure and our notion of record type correponds to a feature structure. One way in which our type system differs from HPSG is that we have both records and record types where HPSG has just feature structures. We will consider a sign to be a record representing a pairing of a speech event and a type representing the content. One advantage of considering a sign as a record rather than a function as in (13) is that there is no directionality in a record as there is in a function. Thus the record can be associated with either interpretation (from speech event to content) or generation (from content to speech event). We can make a straightforward relationship between a function such as (13) and a record type (14).

(14)
$$\begin{bmatrix} \text{s-event} & : & T_{SpEv} \\ \text{cnt=}T_{Cnt} & : & \textit{Cnt} \end{bmatrix}$$

(14) is a type of signs. Notice that the 'cnt'-field in (14) is a manifest field corresponding to the fact that the function in (13) returns the type T_{Cnt}, not an object of type T_{Cnt}. This means that the 'cnt' field in (14) requires that the type itself is in the 'cnt' field in a record of the type, that is, in the sign. The type *Cnt* is the type of contents. For the moment we will say that *Cnt* is the type *RecType*, that is, that contents are record types. This is because, for the moment, we will restrict our attention to declarative sentences. When we come to look at constituents of sentences and speech acts other than assertions we will need to expand *Cnt* to include other kinds of entities as well. Restricting our attention first to complete declarative sentences is similar to starting

with propositional logic before moving on to more complex analysis. The type *Sign* of signs in general is given in (15).

(15)
$$\begin{bmatrix} \text{s-event} & : & \mathit{SEvent} \\ \text{cnt} & : & \mathit{Cnt} \end{bmatrix}$$

A record of this type, a sign, will pair a speech event with a content. We will refine our definition of *Sign* as we progress.

2.3 Information exchange in dialogue

We start by considering simple dialogues such as (16) which might occur between two people one of whom is instructing the other about simple facts or between a user and a system where the user is adding simple facts to a database using a natural lanuage interface.

(16)
User:	Dudamel is a conductor
System:	Aha
User:	Beethoven is a composer
System:	OK

The job of the dialogue partner identified as "System" is to record the facts in memory and confirm to the dialogue partner identified as "User" that this has happened. It seems straightforward to think of the user's utterances in (16) as corresponding to signs as described in Section 2.2. For example, the user's first utterance could be regarded as corresponding to a sign of the type in (17).

(17)
$$\begin{bmatrix} \text{s-event:}\begin{bmatrix} \text{e} & : & \text{"Dudamel is a conductor"} \end{bmatrix} \\ \text{cnt=}\begin{bmatrix} \text{e} & : & \text{conductor(dudamel)} \\ \text{c}_{\text{tns}} & : & \text{final\_align}(\Uparrow\text{s-event.e,e}) \end{bmatrix} \end{bmatrix} : \mathit{RecType}$$

Here "Dudamel is a conductor" is a convenient abbreviation for (18).

(18) $\left[\text{e:"Dudamel"}\right] \frown \left[\text{e:"is"}\right] \frown \left[\text{e:"a"}\right] \frown \left[\text{e:"conductor"}\right]$

where for any word w, "w" is the type of event where w is uttered. "Dudamel is a conductor" is thus a type of string of events of word utterances and is thus a subtype of *Phon*, given our assumptions in Section 2.1.

The content is that Dudamel is a conductor and that his being a conductor is aligned with the speech event in that the speech event occurs simultaneously with the end of the event of Dudamel being a conductor. This is not to say that Dudamel will not continue to be a conductor after the speech event but rather to say that we are aligning the speech event with what has happened so far up to and including the speech event. (The simple present in English in contrast to the present progressive and the simple present in many other languages seems to require this.) How do we align events? We use the technique developed by Fernando (see, for example, Fernando, 2008) of creating a single event which includes both events as a part. We will exploit our record technology to keep track of the separate events in the larger event and to achieve something corresponding to what Fernando calls *superposition*. We might require that the event which is the coordination of the two events of type "Dudamel is a conductor" and 'conductor(dudamel)' is of the type in (19).

(19) $\begin{bmatrix} e_1 & : & \text{“Dudamel is a conductor”} \\ e_2 & : & \text{conductor(dudamel)} \end{bmatrix}$

Another option is to require that the coordinated event type explicitly allow for there to be events of the type 'conductor(dudamel)' prior to the utterance as in (20).

(20) $\left[\begin{array}{lcl} e & : & \text{conductor(dudamel)} \end{array} \right]^{*\frown} \left[\begin{array}{lcl} e & : & \begin{bmatrix} e_1 & : & \text{“Dudamel is a conductor”} \\ e_2 & : & \text{conductor(dudamel)} \end{bmatrix} \end{array} \right]$

Here the dimension 'e' splits into two subdimension 'e.e_1' and 'e.e_2'. If we wish to be explicit about the fact that a situation of type "Dudamel is a conductor" is a string of word utterances we can give the more detail type in (21).

(21) $\left[\begin{array}{lcl} e & : & \text{conductor(dudamel)} \end{array} \right]^{*\frown} \left[\begin{array}{lcl} e & : & \begin{bmatrix} e_1 & : & \text{“Dudamel”} \\ e_2 & : & \text{conductor(dudamel)} \end{bmatrix} \end{array} \right]^{\frown}$
$\left[\begin{array}{lcl} e & : & \begin{bmatrix} e_1 & : & \text{“is”} \\ e_2 & : & \text{conductor(dudamel)} \end{bmatrix} \end{array} \right]^{\frown}$
$\left[\begin{array}{lcl} e & : & \begin{bmatrix} e_1 & : & \text{“a”} \\ e_2 & : & \text{conductor(dudamel)} \end{bmatrix} \end{array} \right]^{\frown}$
$\left[\begin{array}{lcl} e & : & \begin{bmatrix} e_1 & : & \text{“conductor”} \\ e_2 & : & \text{conductor(dudamel)} \end{bmatrix} \end{array} \right]$

This explicitly requires that Dudamel is a conductor during the utterance of each individual word. Both the types (20–21) are facilitated by the fact that 'conductor(dudamel)' is a *state*-type, that is, given a situation e : conductor(dudamel) we can regard it as a string of events of type $\left[\text{e:conductor(dudamel)}\right]^+$. We will return to aspectual types other than state below. The predicate 'final_align' in (17) requires alignment of the speech event and the described event in the way we have exemplified in (20) and (21). The definition of what counts as a witness for final_align(e_1,e_2) given in Appendix A.13 requires that e is of this type just in case e is an event where e_1 is aligned with a final segment of e_2, that is in e there is a split in dimension in the final segment as illustrated in (21). The notation '$\Uparrow$' in (17) indicates that the path 'x' is not to be found in the local record type which is required to be the value of 'cnt' but in the next higher record type with the fields 's-event' and 'cnt'. This notation is explained in Appendix A.11.

This sign type (17) seems to give us what we need in order to explain how an utterance of *Dudamel is a conductor* can convey the information that Dudamel is a conductor. If both dialogue participants have this sign type among their resources then the User knows that in order to convey this content she has to make an utterance which witnesses the appropriate speech event type. The System knows that on observing a speech event of this type the corresponding content should be recorded.

Things are not as straightforward, however, for the acknowledgements *Aha* and *OK* expressed by the system. It is not obvious whether these utterances are to be regarded as signs at all. Certainly a speech event is involved but one might question what content they have. One suggestion would be that the content of *Aha* uttered after an assertion by the other dialogue partner would be the same as the content of that assertion. Thus the system is expressing the same content as the user. This may or may not be true. But such an analysis seems to be missing a central point about what is going on in this dialogue, namely that the user is making an assertion and the system is acknowledging that the content has been accepted and duly processed. In order to account for this kind of fact Ginzburg in a large body of work has developed the notion of a dialogue gameboard, most recently formulated in terms of TTR in Ginzburg (2012); Ginzburg and Fernández (2010). In the computational dialogue systems literature this have given rise to the Information State Update (ISU) approach (Larsson and Traum, 2001; Larsson, 2002) which is also described in Ginzburg and Fernández (2010). In Chapter 1 we introduced the notion of an information state as a record containing a field labelled 'agenda' and used the word "gameboard" to refer to a type of information state. Our aim there was to show that the kind of gameboard analysis introduced for dialogue in this literature is also important for the coordination of joint action by agents in general. The gameboards that have been used for dialogue analysis have a number of fields in addition to the agenda. Each dialogue participant will have among their resources a record type, their dialogue gameboard which represents their understanding of (what Larsson call their take on) their current information state. Following Larsson (2002) we place information which the agent assumes to be common with its interlocutors under the label 'shared' in the gameboard and also have a field with the label 'private' representing information about the state of the dialogue which is not shared with other dialogue participants. This will include, for example, plans for what should be said next represented in the agenda. In Figure 2.1 we give a schematic view of

the gameboards associated with each of the dialogue participants in the first exchange in (16).

This assumes ideal communication. There is lots that could go wrong which could have the consequence that the two agents become misaligned and an important part of this framework is to provide a basis for the description of miscommunication as well as communication. (See Ginzburg (2012) for more discussion of this.)

We treat the dialogue information states represented by the square boxes as records as in (22).

$$(22)\quad \begin{bmatrix} \text{private} & = & \begin{bmatrix} \text{agenda} & = & \textit{AGENDA} \end{bmatrix} \\ \text{shared} & = & \begin{bmatrix} \text{latest-utterance} & = & \textit{L-UTT} \\ \text{commitments} & = & \textit{COMM} \end{bmatrix} \end{bmatrix}$$

What kinds of objects should *AGENDA*, *L-UTT* and *COMM* be? They will be defined with respect to the agent who owns the information state which, for convenience, we will refer to as *SELF*. We will see as we proceed with the discussion below that *SELF* is related to the notion of *de se* type act discussed in Chapter 1.5.

We will say that *AGENDA* is a list of dialogue move types, that is, the types of dialogue moves that *SELF* plans to realize by means of a creation type act. Recall from Chapter 1 that this does not necessarily mean that *SELF* is the main actor in the event realizing the move type. It can for example be a type of move to be carried out by an interlocutor which *SELF* should wait for. This will give us a mechanism for handling basic turn-taking in dialogue. (See Sacks *et al.*, 1974 for the classic work on turn-taking.) We will define a dependent type *Move* such that for any agent A, *Move*(A) is the type of dialogue moves in which A is involved. For now we will say that there are two ways in which an agent can be involved in a dialogue act: as speaker (or performer) or as hearer (part of the audience to whom the dialogue act is addressed).[6] Performing a dialogue move is a *de se* type act of creation as discussed in Chapter 1.5. Being the hearer or audience of a move type involves a *de se* type act of judgement as discussed there. *AGENDA* should thus be a list of move types depending on *SELF* (that is, subtypes of *Move*(*SELF*)). We introduce a dependent type, *MoveType*, such that for any a:*Ind*, T:*MoveType*(a) iff $T \sqsubseteq$ *Move*(a). *AGENDA* is thus a list of move types and will have type [*MoveType*(*SELF*)]. For any type T, $[T]$ is the type of lists all of whose members are of type T (see Appendix A.5). We will come back to the details of *Move* below.

L-UTT should tell us what move (or moves[7]) has just been carried out. But we will need more information than this. We will need information about what (the agent *SELF* thinks) was actually said. For this we will use a chart, i.e. a set of edges between vertices representing hypotheses

[6] A third way of being involved in a dialogue act which we will not take account of here is as an overhearer.

[7] See Larsson (2002) for a proposal where dialogue contributions involve several moves. For now we will make the simplifying assumption that utterances are associated with a single move.

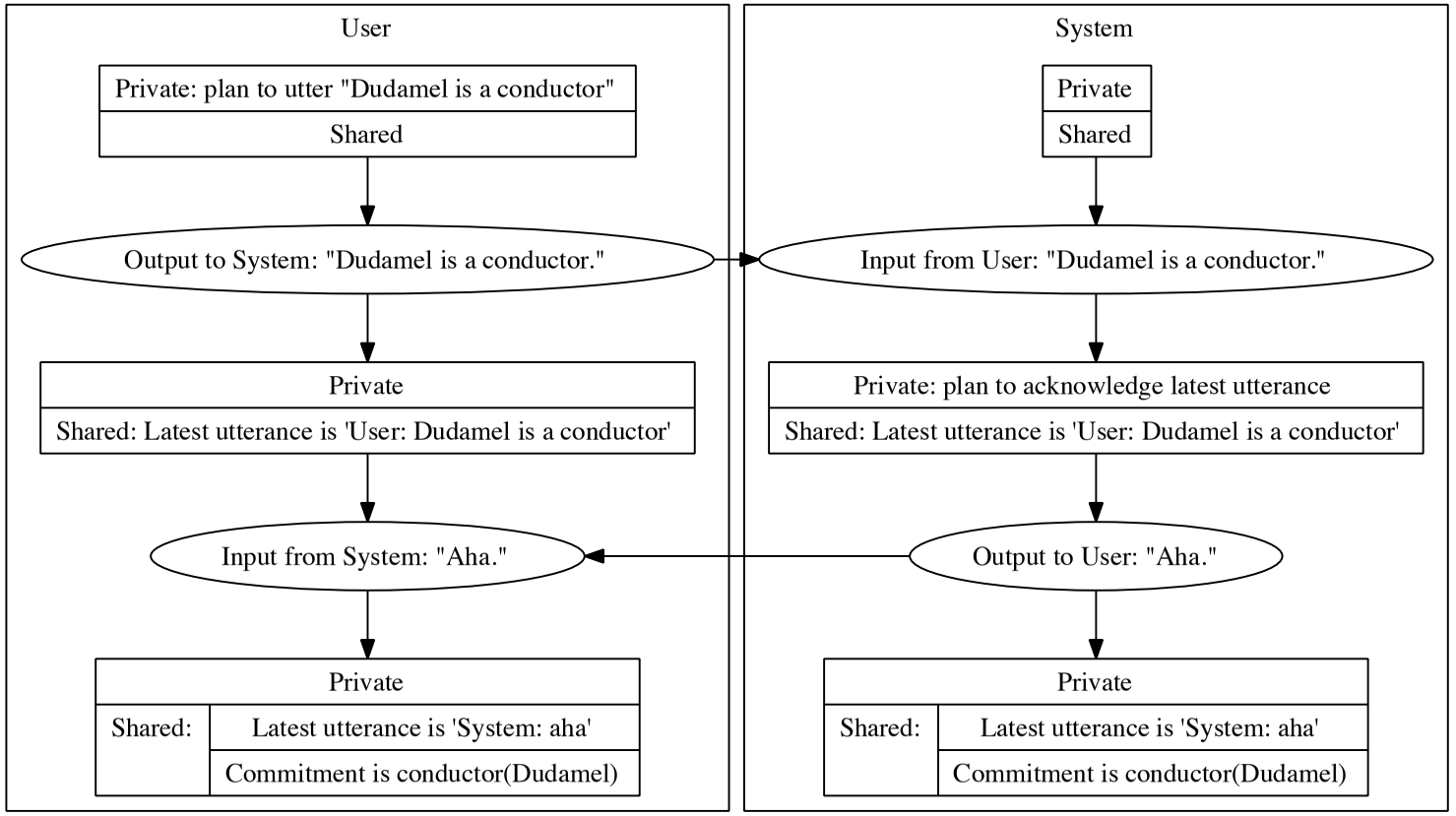


Figure 2.1: Dialogue management:“Dudamel is a conductor”

about parts of the utterance, that is, sign types associated with parts of the utterance. The move should be predictable from the chart by a process of move-interpretation for which we will use the predicate 'm-interp'. Thus *L-UTT* should itself be of the type (23).

(23)
$$\begin{bmatrix} \text{move} & : & \textit{Move}(\textit{SELF}) \\ \text{chart} & : & \textit{Chart} \\ \text{e} & : & \text{m-interp(chart,move)} \end{bmatrix}$$

The type *Chart* we will say more about in Chapter 3.

The commitments field has normally been considered as a set of facts or propositions (Ginzburg, 2012; Larsson, 2002). Here we will treat them as a single record type, i.e. a member of the type *RecType*. Using a single type will make it more straightforward to deal with issues like consistency and anaphora [????].

Thus information states can belong to the type (24).

(24)
$$\begin{bmatrix} \text{private:} \left[\text{agenda:}[\textit{MoveType}(\textit{SELF})]\right] \\ \text{shared:} \begin{bmatrix} \text{latest-utterance:} \left[\begin{matrix}\text{move:}\textit{Move}(\textit{SELF}) \\ \text{chart:}\textit{Chart} \\ \text{e:m-interp(chart,move)}\end{matrix}\right] \\ \text{commitments:}\textit{RecType} \end{bmatrix} \end{bmatrix}$$

(Here, by convention the labels 'chart' and 'move' in $\left[\text{e:m-interp(chart,move)}\right]$ refer to the path down to the minimal record in which the e-field occurs, that is 'shared.latest-utterance.chart' and 'shared.latest-utterance.move' respectively.)

(24) is, however, not quite general enough. It requires that there always will be a latest utterance. At the beginning of a dialogue this will not be the case and we need a way of representing that there is no previous utterance. We will use a special type *Nil* for this. The only object of this type is 'nil'. If you like you can think of 'nil' as the empty set and *Nil* as a type whose only witness is the empty set. At the beginning of a dialogue there will not be any shared commitments either. However, commitments is a record type and record types are sets of fields (see Appendix A.11) including the empty set. Therefore, it will be natural to use the empty record type '[ ]' for the commitments at the beginning of a dialogue. Therefore the only adjustment we need to make to (24) in order to include dialogue initial information states is to the shared.latest-utterance field as in (25).

(25)
$$\left[\begin{array}{l}\text{private:}\left[\text{agenda:[}\textit{MoveType}(\textit{SELF})\text{]}\right] \\ \text{shared:}\left[\begin{array}{l}\text{latest-utterance:}\left[\begin{array}{l}\text{move:}\textit{Move}(\textit{SELF}) \\ \text{chart:}\textit{Chart} \\ \text{e:m-interp(chart,move)}\end{array}\right]\vee\textit{Nil} \\ \text{commitments:}\textit{RecType}\end{array}\right]\end{array}\right]$$

(25) uses a join type (Appendix A.7). For any two types T_1 and T_2 you can form the join (or disjunction) $T_1 \vee T_2$. $a : T_1 \vee T_2$ just in case either $a : T_1$ or $a : T_2$.

We will use notation including '*SELF*' as in (25) to represent types which are derived from dependent types by applying them to the argument represented by *SELF*. This notational convention will save us a good deal of complication in presentation and it is always possible to recover the dependent type from which the type is derived by creating a function which maps an individual to the appropriate type. Thus in the case of (25) the dependent type would be (26).

(26) $\lambda a{:}\textit{Ind}\,.$
$$\left[\begin{array}{l}\text{private:}\left[\text{agenda:[}\textit{MoveType}(a)\text{]}\right] \\ \text{shared:}\left[\begin{array}{l}\text{latest-utterance:}\left[\begin{array}{l}\text{move:}\textit{Move}(a) \\ \text{chart:}\textit{Chart} \\ \text{e:m-interp(chart,move)}\end{array}\right]\vee\textit{Nil} \\ \text{commitments:}\textit{RecType}\end{array}\right]\end{array}\right]$$

Dialogue moves are a type of event in which an actor (normally speaker) is related to an intended audience, an illocutionary force (such as 'assert') and a content (that is, for our present purposes, a record type such as $\left[\text{e:conductor(Dudamel)}\right]$).

We will take dialogue moves to be a pairing of speech acts and content. The type of speech acts (*SpeechAct*) will be taken to be a subtype of the type of speech events (*SEvent*) as defined in (10) on p. 37. In particular this will mean that there is a field in a speech act for the speaker (labelled by 'sp') and another for the audience (labelled by 'au'). More specifically we will take the type *Move*(a) to be an abbreviation for (27).

(27) $\left[\text{e:}\textit{SpeechAct}\right] \wedge \left(\ \left[\text{e:}\left[\text{sp=}a{:}\textit{Ind}\right]\right] \vee \left[\text{e:}\left[\text{au=}a{:}\textit{Ind}\right]\right]\ \right) \wedge \textit{MoveContent}$

The type in (27) is a meet type (Appendix A.8).[8] If T_1 and T_2 are types, then an object a is of type $T_1 \wedge T_2$ just in case $a : T_1$ and $a : T_2$.

[8] Strictly speaking, this type should be written with parentheses since we are assuming a binary meet operation: $(\left[\text{e:}\textit{SpeechAct}\right] \wedge ((\ \left[\text{e:}\left[\text{sp=}a{:}\textit{Ind}\right]\right] \vee \left[\text{e:}\left[\text{au=}a{:}\textit{Ind}\right]\right]\ ) \wedge \textit{MoveContent}))$ but we will often omit parentheses for clarity.

Note that (27) requires a to be either the speaker or the audience of the speech act and does not rule out the possibility that a is both speaker and audience (i.e. a is talking to herself).

We will not attempt a complete inventory of speech act types here. Preliminarily, we could define *SpeechAct* to be

Assertion∨*Query*∨*Command*∨*Acknowledgement*,

that is, a join type (Appendix A.7) of all the available speech act types.[9] Something will be of this type just in case it is of at least one of the types of the join. Each of the speech act types are subtypes of *SEvent* and can be defined as in (28).

(28)

$$\begin{array}{lll} \textit{Assertion} & - & \textit{SEvent} \mathbin{\dot{\wedge}} \begin{bmatrix} \text{e:}\textit{Phon} \\ \text{c}_{\text{illoc}}\text{:assertion(e)} \end{bmatrix} \\ \textit{Query} & - & \textit{SEvent} \mathbin{\dot{\wedge}} \begin{bmatrix} \text{e:}\textit{Phon} \\ \text{c}_{\text{illoc}}\text{:query(e)} \end{bmatrix} \\ \textit{Command} & - & \textit{SEvent} \mathbin{\dot{\wedge}} \begin{bmatrix} \text{e:}\textit{Phon} \\ \text{c}_{\text{illoc}}\text{:command(e)} \end{bmatrix} \\ \textit{Acknowledgement} & - & \textit{SEvent} \mathbin{\dot{\wedge}} \begin{bmatrix} \text{e:}\textit{Phon} \\ \text{c}_{\text{illoc}}\text{:acknowledgement(e)} \end{bmatrix} \end{array}$$

Here the subscript 'illoc' stands for "illocutionary" indicating that the condition provides information about the illocutionary force of the speech act. The symbol $\dot{\wedge}$ represents the merge operation defined in Appendix A.12. In (28) the relevant merges will be the unions of the sets of fields represented by *SEvent* and the type consisting of the 'e' and 'illoc' fields. This is illustrated in (29) for *Assertion*. (29a) (where *SEvent* is spelled out) is identical with (29b).

[9] Strictly speaking, this type should be written with parentheses since we are assuming a binary join operation: (*Assertion*∨(*Query*∨(*Command*∨*Acknowledgement*))) but we will often omit parentheses for clarity.

(29)

a.
$$\begin{bmatrix} \text{e-loc} & : & \textit{Loc} \\ \text{sp} & : & \textit{Ind} \\ \text{au} & : & \textit{Ind} \\ \text{e} & : & \textit{Phon} \\ \text{c}_{\text{loc}} & : & \text{loc(e,e-loc)} \\ \text{c}_{\text{sp}} & : & \text{speaker(e,sp)} \\ \text{c}_{\text{au}} & : & \text{audience(e,au)} \end{bmatrix} \wedge\!\!\!\cdot \begin{bmatrix} \text{e:}\textit{Phon} \\ \text{c}_{\text{illoc}}\text{:assertion(e)} \end{bmatrix}$$

b.
$$\begin{bmatrix} \text{e-loc} & : & \textit{Loc} \\ \text{sp} & : & \textit{Ind} \\ \text{au} & : & \textit{Ind} \\ \text{e} & : & \textit{Phon} \\ \text{c}_{\text{loc}} & : & \text{loc(e,e-loc)} \\ \text{c}_{\text{sp}} & : & \text{speaker(e,sp)} \\ \text{c}_{\text{au}} & : & \text{audience(e,au)} \\ \text{c}_{\text{illoc}} & : & \text{assertion(e)} \end{bmatrix}$$

Finally, the type *MoveContent* in (27) relates the type of the content of the move to the type of the move. We define it preliminarily as the join type in (30).

(30)
$$\begin{bmatrix} \text{e} & : & \textit{Assertion} \\ \text{cnt} & : & \textit{RecType} \\ \text{c}_{\text{cnt}} & : & \text{content(e,cnt)} \end{bmatrix} \vee \begin{bmatrix} \text{e} & : & \textit{Query} \\ \text{cnt} & : & \textit{Question} \\ \text{c}_{\text{cnt}} & : & \text{content(e,cnt)} \end{bmatrix} \vee \begin{bmatrix} \text{e} & : & \textit{Command} \\ \text{cnt} & : & \textit{RecType} \\ \text{c}_{\text{cnt}} & : & \text{content(e,cnt)} \end{bmatrix} \vee \begin{bmatrix} \text{e} & : & \textit{Acknowledgement} \end{bmatrix}$$

Note that this allows for acknowledgements such as *ok* not to have any content (although it does not prevent them from having content). We will return later to discussion of whether this is a reasonable claim for acknolwedgements, while noting that this would be one way of dealing with "phatic" communication such as greetings like *Hello*.

We will be able to read partial information about a dialogue move from certain aspects of a speech-event. For example, an utterance of *ok* may tell us that the dialogue move is of type (31).

(31)
$$\begin{bmatrix} \text{e} & : & \textit{Acknowledgement} \\ \text{sp=}\textit{SELF} & : & \textit{Ind} \end{bmatrix}$$

If an utterance of *ok* is to have content after all, we will have to look to the previous utterance to find it. An utterance of *Dudamel is a conductor* may give us information about the type of speech act (*Assertion*) and the content ($\left[\text{e:conductor(Dudamel)}\right]$). Note, however, that we only get such a fully specified content if we have a unique individual 'Dudamel' whom we associate with utterances of the name *Dudamel*. If the resources we have available do not give us such an individual associated with *Dudamel* then we only get the information that somebody named Dudamel is a conductor, that is, we may only get partial information about the content the speaker intended to communicate. Consider the example *Strauss is a composer*. There are at least two famous composers named Strauss (and also some more not so famous ones). If our available resources give us two people associated with the name *Strauss* we will not know which of them is being referred to. Representing contents as record types will enable us to handle this content underspecification. However, in order to do this we will need to abandon our current simplifying "propositional logic" assumption that sentences come as unanalyzed wholes associated with their contents. This we will do in Chapter 3.

Not surprisingly, when we are dealing with an agenda as in (25), a plan for future action, we have got ourselves into a situation where we need types rather than the objects. The things that are on the agenda list are not actual events, but rather *types* of events planned for the future. Normally the types occurring on the agenda will be subtypes of *Move*(*SELF*), though we may wish to include types of events like looking something up in a database, i.e. non-speech events.

For the most part types on the agenda will not be completely specified types. That is they will not be types all of whose fields are manifest (restricted to particular objects of those types). Frequently it will be the case that we specify the content of the move but leave open the phonology, that is, the type will specify the content of what is to be said but not actually what is to be said or even perhaps which language it should. We want, for example, to be able to say that a speaker is carrying out the same type of move independently of which language they are speaking. Thus, if the user says to the system "Dudamel is a conductor" or (in Swedish) "Dudamel är dirigent" she will in both cases have carried out a move involving the assertion of the content $\left[\text{e:conductor(Dudamel)}\right]$. This abstraction will be important, for example, if we want to change language in the middle of a dialogue, as people sometimes do.[10] At the same time it is the normal case to continue a dialogue in the same language and thus we need to note which language was used in the previous utterance, i.e. keep track of what was actually said. This information will be in the chart which is part of the latest move. In the chart there will be more information about what was actually said which will be important when it comes to dealing with parts of the utterance for things like clarification and anaphora. But this again requires us to abandon our current simplifying "propositional logic" assumption.

We will assume that agents do not have complete information about the information state, that is, they reason in terms of *types* of information state (that is, gameboards). The basic intuition

[10]This phenomenon is known as code-switching (Bullock and Toribio, 2009).

behind our reasoning about information state updates can be expressed as in (32).

(32) If $r_i : T_i$, then $r_{i+1} : T_{i+1}(r_i)$

That is, given that we believe that the current information state is of type T_i (recall that we can come to this belief without having any belief about which specific information state is involved), then we can conclude that the next information state is of type T_{i+1} which can depend on the current information state. According to this, we can have a hypothesis about the type of the next information state even though we may not know exactly what the current information state is. Exactly which type the next information state belongs to depends, though, on the exact nature of the current information state. Thus the dependency in our types provides us with an additional means for representing underspecification.

This basic rule of inference corresponds to a function from records to record types, a function of type ($T_i \rightarrow$ *RecType*), that is, one kind of update function we were using in Chapter 1. Such a function is of the form (33).

(33) $\lambda r\!:\!T_i \,.\, T_{i+1}(r)$

Things are a litte more complicated than this, however, because this only represents the change from one information state to another, whereas in fact this change is triggered by a speech event which bears an appropriate relation to the current information state represented by r. Thus we are actually interested in functions from the current information state to a function from events to the new information state, as in (34).

(34) $\lambda r\!:\!T_i \,.\, \lambda e\!:\!T_e(r) \,.\, T_{i+1}(r, e)$

This is the other kind of update function we were using in Chapter 1.[11] Let us consider the update function which the user could use in order to update her information state after her own utterance of *Dudamel is a conductor*. This is modelled on the kind of integration rules discussed

[11]This is one of a number of ways of characterizing update in this kind of framework. One might for instance think of the type of the speech event as being part of the current information state. Also instead of using an update function one can use a record type with a 'preconditions'-field and an 'effect'-field. Both Ginzburg (2012) and Larsson (2002) have this kind of approach.

in Larsson (2002).

(35)

$$\lambda r{:}\left[\text{private} : \left[\text{agenda} : {}_{ne}[\mathit{MoveType}(\mathit{SELF})]\right]\right]$$
$$\lambda u{:}\left[\begin{array}{lll}\text{move} & : & \text{fst}(r.\text{private.agenda}) \mathbin{\dot{\wedge}} \left[\text{e:}\begin{bmatrix}\text{sp=}\mathit{SELF}\text{:}\mathit{Ind}\\ \text{au:}\mathit{Ind}\end{bmatrix}\right] \mathbin{\dot{\wedge}} \left[\text{e:}\mathit{Assertion}\right] \\ \text{chart} & : & \mathit{Chart} \\ \text{e} & : & \text{m-interp(chart,move)}\end{array}\right].$$
$$\left[\begin{array}{l}\text{private:}\left[\text{agenda=}\left[\begin{array}{l}\text{e:}\mathit{Acknowledgement} \mathbin{\dot{\wedge}} \begin{bmatrix}\text{sp=}u.\text{move.e.au:}\mathit{Ind}\\ \text{au=}\mathit{SELF}\text{:}\mathit{Ind}\end{bmatrix} \\ \text{cnt=}u.\text{move.cnt:}\mathit{RecType} \\ \text{c}_{\text{cnt}}\text{:content(e,cnt)} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \text{rst}(r.\text{private.agenda})\end{array}\right] :[\mathit{MoveType}(\mathit{SELF})]\right] \\ \text{shared:}\left[\text{latest-utterance:}\begin{bmatrix}\text{move=}u.\text{move:}\mathit{Move}(\mathit{SELF})\\ \text{chart=}u.\text{chart:}\mathit{Chart}\\ \text{e=}u.\text{e:m-interp(chart,move)}\end{bmatrix}\right]\end{array}\right]$$

This function maps information states (records), r, which have a non-empty agenda to a function that maps events to a type of information state. (See Appendix A.5 for an account of non-empty list types.) It thus requires that the current information state (the first argument to the function) have a non-empty agenda. The second argument to the function (represented by u) requires the move associated with the speech-event to be of the first type on the agenda in r, the current information state, and also to be an assertion with *SELF* as the speaker (see Appendix A.8 for a discussion of meet types, that is, conjunction). It also requires that the chart associated with this utterance can be interpreted as a move of that type. The requirements on the arguments to the function represent the preconditions. The type that results from applying the function to its arguments represents the effect of the update. This type requires the agenda to be result of replacing the first type on the agenda in r with an acknowledgement where the speaker is the audience of the assertion move and the audience of the acknolwedgement is *SELF*. The content of the acknowledgement is the same as the content of the assertion. That is, what is being acknowledged is the content of the assertion. It furthermore requires the latest-utterance field to contain the move and chart of the utterance u. The idea is that this function should be used to predict the type of the next information state on the basis of the current information state and the observed event. That is, if we believe the current information state to be of the domain type of the update function and we observe an event of the required type then we reason that the updated information state should be of the type resulting from applying the function to the current information state. Thus this update function will be used in the same way as the update functions we discussed in Chapter 1. However, the gameboards involved are now more complex.

We will now examine how such an update function could be used to reason about an update. Let

us suppose that the user considers the current information state to be of type:

(36)
$$\begin{bmatrix} \text{private:} \begin{bmatrix} \text{agenda=[} \begin{bmatrix} \text{e:}\textit{Assertion} \dot{\wedge} \left[\text{sp=}\textit{SELF}\text{:}\textit{Ind}\right] \\ \text{cnt=}\left[\text{e:conductor(dudamel)}\right]\text{:}\textit{RecType} \\ \text{c}_{\text{cnt}}\text{:content(e,cnt)} \end{bmatrix} \text{] :[}\textit{RecType}\text{]} \end{bmatrix} \\ \text{shared:} \begin{bmatrix} \text{latest-utterance:}\textit{Nil} \\ \text{commitments=[]:}\textit{RecType} \end{bmatrix} \end{bmatrix}$$

This represents that the user intends to assert that Dudamel is a conductor represented by the record type $\left[\text{e:conductor(Dudamel)}\right]$. The user also believes that there was no previous utterance and no commitments, i.e. that the planned utterance will be dialogue initial.

Suppose now that the user utters *Dudamel is a conductor* and judges this utterance event u_1 to be an event of type (37).

(37)
$$\begin{bmatrix} \text{move} & : & \begin{bmatrix} \text{e:}\textit{Assertion} \dot{\wedge} \left[\text{sp=}\textit{SELF}\text{:}\textit{Ind}\right] \\ \text{cnt=}\left[\text{e:conductor(Dudamel)}\right]\text{:}\textit{RecType} \\ \text{c}_{\text{cnt}}\text{:content(e,cnt)} \end{bmatrix} \\ \text{chart} & : & \textit{Chart} \\ \text{e} & : & \text{m-interp(chart,move)} \end{bmatrix}$$

The user will have more information about the nature of the chart (that is, about what was actually said and how it might be analyzed) than we have represented but we will leave this underspecified for now.

Clearly in the user's judgement the utterance u_1 fulfils the requirements placed on it by (35) since the move interpretation associated with it is of the type which occurs at the head of the agenda. Note that we are reasoning with this function without actually providing it with an argument since we only have a (hypothesized) type of the current information state, not the actual information state. The crucial judgement is that the type of the current information state is a subtype of the domain type of the function. This is sufficient to allow us to come to a conclusion about the type of the new information state.

According to the update function the next information state must be of the type (38).

(38)
$$\left[\begin{array}{l}\text{private:}\left[\text{agenda=[}\left[\begin{array}{l}\text{e:}\textit{Acknowledgement}\wedge_{\cdot}\left[\begin{array}{l}\text{sp=}u_1\text{.move.e.au:}\textit{Ind}\\ \text{au=}\textit{SELF}\text{:}\textit{Ind}\end{array}\right]\\ \text{cnt=}u_1\text{.move.cnt:}\textit{RecType}\\ \text{c}_{\text{cnt}}\text{:content(e,cnt)}\end{array}\right]\text{]:[}\textit{RecType}\text{]}\right]\\ \text{shared:}\left[\text{latest-utterance:}\left[\begin{array}{l}\text{move=}u_1\text{.move:}\textit{Move}\\ \text{chart=}u_1\text{.chart:}\textit{Chart}\\ \text{e=}u_1\text{.e:m-interp(chart,move)}\end{array}\right]\right]\end{array}\right]$$

But we know more about the new information state than what is expressed by the type which results from the update function. Everything we know about the current information state which remains unchanged by the function must be carried over from the current information state. This is related to the frame problem introduced by McCarthy and Hayes (1969).[12] We handle this performing an *asymmetric merge* (see Appendix A.12) of the type we have for the current information state with the type resulting from the update function. The asymmetric merge of two types T_1 and T_2 is represented by $T_1 \boxed{\wedge_{\cdot}} T_2$. If one or both of T_1 and T_2 are non-record types then $T_1 \boxed{\wedge_{\cdot}} T_2$ will be T_2. If they are both record types, then for any label ℓ which occurs in both T_1 and T_2, $T_1 \boxed{\wedge_{\cdot}} T_2$ will contain a field labelled ℓ with the type resulting from the asymmetric merge of the corresponding types in the ℓ-fields of the two types (in order). For labels which do not occur in both types, $T_1 \boxed{\wedge_{\cdot}} T_2$ will contain the fields from T_1 and T_2 unchanged. In this informal statement we have ignored complications that arise concerning dependent types in record types. This is discussed in Appendix A.12. Our notion of asymmetric merge is related to the notion of priority unification (Shieber, 1986).

Let us see how this works with our example. We have assumed that the type under consideration for the current information state, T_{curr}, is (36) and computed that the predicted type of the updated information state, T_{pr}, is (38). Therefore we need to compute $T_{curr} \boxed{\wedge_{\cdot}} T_{pr}$, that is, (39).

[12] For a recent overview of the frame problem see Shanahan (2009).

(39)
$$\begin{bmatrix}\text{private:}\begin{bmatrix}\text{agenda=[}\begin{bmatrix}\text{e:}\textit{Assertion}\dot{\wedge}\left[\text{sp=}\textit{SELF}\text{:}\textit{Ind}\right]\\ \text{cnt=}\left[\text{e:conductor(Dudamel)}\right]\text{:}\textit{RecType}\\ \text{c}_{\text{cnt}}\text{:content(e,cnt)}\end{bmatrix}\text{]:[}\textit{Move}\text{(SELF)]}\end{bmatrix}\\ \text{shared:}\begin{bmatrix}\text{latest-utterance:}\textit{Nil}\\ \text{commitments=[]:}\textit{RecType}\end{bmatrix}\end{bmatrix}$$

$$\boxed{\dot{\wedge}}\begin{bmatrix}\text{private:}\begin{bmatrix}\text{agenda=[}\begin{bmatrix}\text{e:}\textit{Acknowledgement}\dot{\wedge}\begin{bmatrix}\text{sp=}u_1\text{.move.e.au:}\textit{Ind}\\ \text{au=}\textit{SELF}\text{:}\textit{Ind}\end{bmatrix}\\ \text{cnt=}u_1\text{.move.cnt:}\textit{RecType}\\ \text{c}_{\text{cnt}}\text{:content(e,cnt)}\end{bmatrix}\text{]:[}\textit{RecType}\text{]}\end{bmatrix}\\ \text{shared:}\begin{bmatrix}\text{latest-utterance:}\begin{bmatrix}\text{move=}u_1\text{.move:}\textit{Move}\\ \text{chart=}u_1\text{.chart:}\textit{Chart}\\ \text{e=}u_1\text{.e:m-interp(chart,move)}\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

A straightforward way to think of the asymmetric merge of two record types is in terms of the paths in each of them. Both T_{curr} and T_{pr} contain paths 'private.agenda'. The types at the end of the respective paths, however, are distinct singleton types. (Recall that manifest fields $\left[\ell{=}a{:}T\right]$ are a convenient notation for $\left[\ell{:}T_a\right]$ where T_a is a restriction of the type T whose only witness is a.) Therefore we include the complete path from the second type in the result of the asymmetric merge. In the case of the path 'shared.latest-utterance' we have the non-record type *Nil* compared with a record type in T_{pr} and therefore we choose the record type in the result since that is what occurs in the second argument of the asymmetric merge operation. Finally, the path 'shared.commitments' occurs in the first type but not in the second and therefore it occurs in its form from the first type in the result of the asymmetric merge. The result is given in (40) which represents the type of the new information state which has been computed as a result of the update.

(40)
$$\begin{bmatrix}\text{private:}\begin{bmatrix}\text{agenda=[}\begin{bmatrix}\text{e:}\textit{Acknowledgement}\dot{\wedge}\begin{bmatrix}\text{sp=}u_1\text{.move.e.au:}\textit{Ind}\\ \text{au=}\textit{SELF}\text{:}\textit{Ind}\end{bmatrix}\\ \text{cnt=}u_1\text{.move.cnt:}\textit{RecType}\\ \text{c}_{\text{cnt}}\text{:content(e,cnt)}\end{bmatrix}\text{]:[}\textit{RecType}\text{]}\end{bmatrix}\\ \text{shared:}\begin{bmatrix}\text{latest-utterance:}\begin{bmatrix}\text{move=}u_1\text{.move:}\textit{Move}\\ \text{chart=}u_1\text{.chart:}\textit{Chart}\\ \text{e=}u_1\text{.e:m-interp(chart,move)}\end{bmatrix}\\ \text{commitments=[]:}\textit{RecType}\end{bmatrix}\end{bmatrix}$$

Why has the field 'shared.commitments' not been updated after the user has asserted that Dudamel is a conductor? This is because the audience has not yet confirmed that they have understood and accepted the move. We assume that our agents are *cautious* and do not assume that commitments are shared until the dialogue participant(s) they are addressing have confirmed acceptance. This interaction is known as grounding and is discussed (among other places) in Traum (1994) and Larsson (2002).

We shall call the update function (35) **IntegrateOwnAssertion** following the style of Larsson (2002) although this does not correspond exactly to any of Larsson's particular update rules. This then can be used to account for the state that the user is in after asserting that Dudamel is a conductor.

We now need an update function that will account for the effect of this utterance on another dialogue participant. For this we will define a function **IntegrateOtherAssertion** which allows an agent to integrate a move which it perceives to be an assertion.

(41) $\lambda r{:}\left[\text{ private : } \left[\text{ agenda : } [\textit{RecType}]\ \right]\ \right]$

$$\lambda u{:}\left[\begin{array}{l}\text{move:}\left[\begin{array}{l}\text{e:}\textit{Assertion}\dot{\wedge}\left[\begin{array}{l}\text{sp:}\textit{Ind}\\ \text{au=}\textit{SELF}\text{:}\textit{Ind}\end{array}\right]\\ \text{cnt:}\textit{RecType}\\ \text{c}_{\text{cnt}}\text{:content(e,cnt)}\end{array}\right]\\ \text{chart:}\textit{Chart}\\ \text{e:m-interp(chart,move)}\end{array}\right].$$

$$\left[\begin{array}{l}\text{private:}\left[\text{agenda=}\left[\begin{array}{l}\text{e:}\textit{Acknowledgement}\dot{\wedge}\left[\begin{array}{l}\text{sp=}\textit{SELF}\text{:}\textit{Ind}\\ \text{au=}u\text{.move.e.sp:}\textit{Ind}\end{array}\right]\\ \text{cnt=}u\text{.move.cnt:}\textit{RecType}\\ \text{c}_{\text{cnt}}\text{:content(e,cnt)}\end{array}\right]\right.\right.\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\left.\left.\mid r\text{.private.agenda}\right]\ :[\textit{RecType}]\right]\\ \text{shared:}\left[\text{latest-utterance:}\left[\begin{array}{l}\text{move=}u\text{.move:}\textit{Move}\\ \text{chart=}u\text{.chart:}\textit{Chart}\\ \text{e=}u\text{.e:m-interp(chart,move)}\end{array}\right]\right]\end{array}\right]$$

If an agent uses (41) to update then the new information state will contain a move type on the agenda which involves acknowledging the content of the assertion by the other dialogue partner. This update function is also cautious in that it does not yet update the shared commitments since the acknowledgement is only scheduled on the agenda but has not yet been performed.

If an agent performs an acknowledge-event ("ok") and it can integrate it with the update function **IntegrateOwnAcknowledgement** which will finally perform an update of shared.commitments. Before we define this update function we will examine what needs to happen in order to update the commitments.

Suppose that in the dialogue so far it has been established that Dudamel is a conductor and that this is represented by the record type (42).

(42) $\left[\text{ e : conductor(Dudamel) }\right]$

Suppose further that the latest move has the content that Beethoven is a composer, namely (43).

(43) $\left[\ \text{e} \ : \ \text{composer(Beethoven)}\ \right]$

One obvious way to combine them would be to merge them, that is, (44a) which is identical with (44b) which in turn is identical with (44c), given the definition in Appendix A.12 which requires that the merge of any two types which are not both record types is identical with the meet of the two types.

(44) a. $\left[\ \text{e} \ : \ \text{conductor(Dudamel)}\ \right] \dot{\wedge} \left[\ \text{e} \ : \ \text{composer(Beethoven)}\ \right]$

b. $\left[\ \text{e} \ : \ \text{conductor(Dudamel)} \dot{\wedge} \text{composer(Beethoven)}\ \right]$

c. $\left[\ \text{e} \ : \ \text{conductor(Dudamel)} \wedge \text{composer(Beethoven)}\ \right]$

For the simple storing of information represented by predicates and names represented in (16) this might be sufficient. It makes the claim that all the information is collected into one eventuality. In more narrative dialogues referring to separate events which we may wish to be able to refer back to this would be an inadequate solution, however. It would be better if we have a way of keeping the labels 'e' separate so that they don't clash, for example in (45a) which is identical with (45b)

(45) a. $\left[\ \text{e}_1 \ : \ \text{conductor(Dudamel)}\ \right] \dot{\wedge} \left[\ \text{e}_2 \ : \ \text{composer(Beethoven)}\ \right]$

b. $\left[\begin{array}{ll} \text{e}_1 & : \ \text{conductor(Dudamel)} \\ \text{e}_2 & : \ \text{composer(Beethoven)} \end{array}\right]$

The potential problems of label clash become very clear if we consider the types in (46a) corresponding to *a boy hugged a dog* and *a girl stroked a cat*. (46a) is identical with (46b) and has a single individual which is both a girl and a boy stroking another individual which is both a dog and a cat.

(46) a.
$$\begin{bmatrix} x & : & \textit{Ind} \\ c_{boy} & : & boy(x) \\ y & : & \textit{Ind} \\ c_{dog} & : & dog(y) \\ e & : & hug(x,y) \end{bmatrix} \dot{\wedge} \begin{bmatrix} x & : & \textit{Ind} \\ c_{girl} & : & girl(x) \\ y & : & \textit{Ind} \\ c_{cat} & : & cat(y) \\ e & : & stroke(x,y) \end{bmatrix}$$

b.
$$\begin{bmatrix} x & : & \textit{Ind} \\ c_{boy} & : & boy(x) \\ c_{girl} & : & girl(x) \\ y & : & \textit{Ind} \\ c_{dog} & : & dog(y) \\ c_{cat} & : & cat(y) \\ e & : & hug(x,y) \wedge stroke(x,y) \end{bmatrix}$$

One way to get around this problem is to ensure that whenever you introduce new types you always use fresh labels that have not been used before and then use explicit constraints to require identity in cases where it is required. However, when we come to examine compositional semantics in Chapter 3 we will see that it is quite important to refer to particular labels in our rules of combination. Instead of introducing unique *labels* we will use the power of records to introduce unique *paths* when contents are combined. We will use the label 'prev' ("previous"). If T_{old} is the content so far and T_{new} is the content we wish to add then the new combined content will be as in (47a). Thus adding the content of *a girl stroked a cat* to that of *a boy hugged a dog* will yield (47b).

(47) a.
$$\begin{bmatrix} \text{prev} & : & T_{\text{old}} \end{bmatrix} \dot{\wedge} T_{\text{new}}$$

b.
$$\begin{bmatrix} \text{prev} & : & \begin{bmatrix} x & : & \textit{Ind} \\ c_{boy} & : & boy(x) \\ y & : & \textit{Ind} \\ c_{dog} & : & dog(y) \\ e & : & hug(x,y) \end{bmatrix} \\ x & : & \textit{Ind} \\ c_{girl} & : & girl(x) \\ y & : & \textit{Ind} \\ c_{cat} & : & cat(y) \\ e & : & stroke(x,y) \end{bmatrix}$$

In the case of our example with Dudamel and Beethoven the result will be (48).

(48)
$$\begin{bmatrix} \text{prev} & : & \begin{bmatrix} e & : & \text{conductor(Dudamel)} \end{bmatrix} \\ e & : & \text{composer(Beethoven)} \end{bmatrix}$$

If we add a further fact to this, say, that Uchida is a pianist we would obtain (49)

$$(49)\quad \begin{bmatrix} \text{prev} : \begin{bmatrix} \text{prev} : [\ \text{e} : \text{conductor(Dudamel)}\ ] \\ \text{e} : \text{composer(Beethoven)} \end{bmatrix} \\ \text{e} : \text{pianist(Uchida)} \end{bmatrix}$$

This means that we now have to add additional information if we want to require identity, for example if we want the Beethoven and Uchida eventualities (prev.e and e in (49)) to be identical. We will return to these matters when we deal with anaphora in Chapter 3. Note that this strategy also gives us a straightforward record of the order in which content was added.

The update function **IntegrateOwnAcknowledgement** is given in (50).

$$(50)\quad \lambda r{:}\begin{bmatrix} \text{private} : [\ \text{agenda} : {}_{ne}[\textit{RecType}]\ ] \\ \text{shared} : \begin{bmatrix} \text{latest-utterance} : [\ \text{move} : [\ \text{content} : \textit{RecType}\ ]\ ] \\ \text{commitments} : \textit{RecType} \end{bmatrix} \end{bmatrix}$$
$$\lambda u{:}\begin{bmatrix} \text{move} : \text{fst}(r.\text{private.agenda}) \dot{\wedge} [\text{e}{:}\textit{Acknowledgement}] \dot{\wedge} [\text{e}{:}[\text{sp}{=}\textit{SELF}{:}\textit{Ind}]] \\ \text{chart} : \textit{Chart} \\ \text{e} : \text{m-interp(chart,move)} \end{bmatrix}.$$
$$\begin{bmatrix} \text{private}{:}[\text{agenda}{=}\text{rst}(r.\text{private.agenda}){:}[\textit{RecType}]] \\ \text{shared}{:}\begin{bmatrix} \text{latest-utterance}{:}\begin{bmatrix} \text{move}{=}u.\text{move}{:}\textit{Move} \\ \text{chart}{=}u.\text{chart}{:}\textit{Chart} \\ \text{e}{=}u.\text{e}{:}\text{m-interp(chart,move)} \end{bmatrix} \\ \text{commitments}{=}[\text{prev}{:}r.\text{commitments}] \dot{\wedge} u.\text{move.cnt}{:}\textit{RecType} \end{bmatrix} \end{bmatrix}$$

This function will

1. update the agenda with the result of removing the first item on the agenda in r, the information state prior to update
2. update the latest utterance with the current utterance (e.g. the utterance of *ok*)
3. update the commitments to be the result of placing the commitments of r under the label 'prev' and merging with the content of the move in the acknolwedgement, u, (which by the update function **IntegrateOtherAssertion** will be the content of the previous assertion, e.g. the utterance of *Dudamel is a conductor*)

We then need an update function **IntegrateOtherAcknowledgement** which is like **IntegrateOwnAcknowledgment** except that it requires that the move event is directed towards the agent doing the updating. This is given in (51).

(51) $\lambda r{:}\left[\begin{array}{lcl}\text{private} & : & \left[\begin{array}{lcl}\text{agenda} & : & {}_{ne}[\textit{RecType}]\end{array}\right] \\ \text{shared} & : & \left[\begin{array}{lcl}\text{latest-utterance} & : & \left[\begin{array}{lcl}\text{move} & : & \left[\begin{array}{lcl}\text{content} & : & \textit{RecType}\end{array}\right]\end{array}\right] \\ \text{commitments} & : & \textit{RecType}\end{array}\right]\end{array}\right]$
$\lambda u{:}\left[\begin{array}{lcl}\text{move} & : & \text{fst}(r.\text{private.agenda}) \dot{\wedge} \left[\text{e:}\textit{Acknowledgement}\right] \dot{\wedge} \left[\text{e:}\left[\text{au=}\textit{SELF}\text{:}\textit{Ind}\right]\right] \\ \text{chart} & : & \textit{Chart} \\ \text{e} & : & \text{m-interp(chart,move)}\end{array}\right]$.
$\left[\begin{array}{l}\text{private:}\left[\text{agenda=rst}(r.\text{private.agenda})\text{:}[\textit{RecType}]\right] \\ \text{shared:}\left[\begin{array}{l}\text{latest-utterance:}\left[\begin{array}{l}\text{move=}u.\text{move:}\textit{Move} \\ \text{chart=}u.\text{chart:}\textit{Chart} \\ \text{e=}u.\text{e:m-interp(chart,move)}\end{array}\right] \\ \text{commitments=}\left[\text{prev:}r.\text{commitments}\right] \dot{\wedge} u.\text{move.cnt:}\textit{RecType}\end{array}\right]\end{array}\right]$

We have so far talked of update functions in this chapter, functions which given an information state and an utterance will return a type for an updated information state. Update functions specify something about the state that an agent will be in after the occurrence of a certain type of event. We have not, however, specified what it is that will specify that an agent should carry out an action which gives rise to an event of the appropriate type. Formally, these will also be functions which map an information state of a given type to a new type, the type of event which the agent is to bring about. Thus they too will be functions from objects to types (or dependent functions). We will call such functions *action functions*. These are associated with creation type acts (Chapter 1.5). We will introduce one such function, **ExecTopAgenda**, which takes an information state with a non-empty agenda and returns a type for a move of that type and a chart which can be interpreted as that move. It is given in (52).

(52) $\lambda r{:}\left[\begin{array}{lcl}\text{private} & : & \left[\begin{array}{lcl}\text{agenda} & : & {}_{ne}[\textit{RecType}]\end{array}\right]\end{array}\right]$.
$\left[\begin{array}{lcl}\text{move} & : & \text{fst}(r.\text{private.agenda}) \\ \text{chart} & : & \textit{Chart} \\ \text{e} & : & \text{m-interp(chart,move)}\end{array}\right]$

2.4 Resources

While there is no formal distinction between update functions and action functions they are to be used in different ways. Update functions are to be used as instructions to conclude that there is something of the resulting type. Action functions are to be used as instructions to create something of the resulting type. We shall say that they are different kinds of *resources* that are available to an agent. The update and action functions we have discussed in this chapter belong to a general resource for *dialogue management*. We shall see that there are a number of resources which contain both update and action functions and that in general they can be viewed as the two

kinds of enthymemes (inferential and imperative) discussed in Breitholtz' work in progress on Aristotelian enthymemes (Breitholtz and Villing, 2008; Breitholtz, 2010; Breitholtz and Cooper, 2011).

We need more resources: signs and move-interpretations of charts containing signs. In this chapter we are taking signs to be objects of type (15) and the sign corresponding to *Dudamel is a conductor* is (17). For compactness of representation we can define an operation which takes a speech event type and a content and constructs the corresponding sign. This can be defined as in (53).

(53) If σ is a type of speech event and κ is a type (of situation) then

$$\text{sign}(\sigma,\kappa)= \begin{bmatrix} \text{s-event:} \begin{bmatrix} \text{e:}\sigma \end{bmatrix} \\ \text{cnt=} \begin{bmatrix} \text{e:}\kappa \\ \text{c}_{\text{tns}}\text{:final\_align}(\Uparrow\text{s-event.e,e}) \end{bmatrix} \text{:}\mathit{RecType} \end{bmatrix}$$

Note that the operation 'sign' introduces the interpretation of present tense (represented by the field 'c_{tns}'). This is only possible because the resources we are considering concern only simple present tense assertions such as *Dudamel is a conductor*. We will see already in the next chapter that things are not this simple. We can use (53) to create signs types for utterances with specific contents such as *Dudamel is a conductor* or *Beethoven is a composer*. We will use another operation 'sign_{uc}' to create signs with underspecified content as defined in (54).

(54) If σ is a type of speech event then

$$\text{sign}_{uc}(\sigma)= \begin{bmatrix} \text{s-event:} \begin{bmatrix} \text{e:}\sigma \end{bmatrix} \\ \text{cnt:}\mathit{RecType} \end{bmatrix}$$

Now we can characterize the sign types that an agent that can deal with the simple dialogues that we have been characterizing in this chapter as (55).

(55) {sign("Dudamel is a conductor", conductor(dudamel)),
sign("Beethoven is a composer", composer(beethoven)),
sign("Uchida is a pianist", pianist(uchida)),
sign_{uc}("ok"),
sign_{uc}("aha")}

Recall that "Dudamel is a conductor" etc. represent a type of a string of word utterance events. For any word w, "w" is the type of event where w is uttered. For present purposes we assume that the agent has basic types of word utterances as given in (56a). In order to cope with the

content the agent must have a basic type *Ind* to which certain individuals belong as given in (56b). Finally in order to construct the ptypes used for the content the agent would have to have the predicates given in (56c).

(56) a. "Dudamel", "is", "a", "conductor", "Beethoven", "composer", "Uchida", "pianist", "aha", "ok"

b. dudamel, beethoven, uchida : *Ind*

c. predicates with arity $\langle$*Ind*$\rangle$: conductor, composer, pianist

The set of ptypes based on (56b,c) is thus (57).

(57) $\{p(a) \mid p \in \{\text{conductor}, \text{composer}, \text{pianist}\}$ and $a \in \{\text{dudamel}, \text{beethoven}, \text{uchida}\}\}$

Of the ptypes in (57) we could say that 'conductor(dudamel)', 'composer(beethoven)' and 'pianist(uchida)' are non-empty ("true") and the rest are empty, although that may not correspond to the actual facts of the world. (Beethoven was a pianist, for example.) Very often, we are mainly interested in whether a ptype has witnesses (something of the type) or not and not particularly what those witnesses are. In a complete formal treatment, of course, the type system would specify objects which belong to those types. For example, we could say s_1 : conductor(dudamel), s_2 : composer(beethoven) and s_3 : pianist(uchida). Informally, we can say s_1 is a situation where Dudamel is a conductor or which shows that Dudamel is a conductor and so on. The idea of saying that an agent has a certain type in its resources is not so much to say that it has complete information about what belongs to the type (although its memory will contain partial information about what belongs to what types) but rather that it has a way (possibly not entirely decidable) of recognizing an object of the type if it sees one. Thus since I am an agent with the type 'composer(uchida)' in my resources I know (sort of) what it would mean for a situation to be of this type, e.g. a situation in which Uchida has written original musical compositions, had them performed and so on. When we are using our type theory to give an analysis of certain fragments of language we are sometimes interested in going into more detail concerning the criteria for belonging to a given type. Other times we just treat the type as basic and only need to assume that the agent has some way of recognizing objects of the type. It depends on the level of detail we are interested in for the particular analysis.

We have used predicates other than those given in (57) in the types that we have discussed in this chapter. There are "technical" predicates such as 'm-interp' ("move interpretation") which takes as its arguments a chart and a move. If c is a chart and m is a move then m-interp(c,m) will be a non-empty type just in case "m is an interpretation of c". Clearly, this is a case where it is of theoretical interest to us to say more about what constraints this places on c and m.

For the purposes of this chapter, since the parsing involved is a trivial association of strings of words with signs without any constituent analysis, we will equate charts with signs, that is a : *Chart* just in case a : *Sign*. Thus 'm-interp' will relate signs to moves. The definition is given in (58).

(58) a. if c : *Chart* and m : *Move* and for some σ and κ, c=sign(σ,κ), then m-interp(c,m) is non-empty iff m : $\begin{bmatrix} \text{e:}\textit{Assertion} \\ \text{cnt=}c\text{.cnt:}\textit{RecType} \end{bmatrix}$

b. if c : *Chart* and m : *Move* and for some σ, c=sign$_{uc}$(σ), then m-interp(c,m) is non-empty iff m : $\begin{bmatrix} \text{e:}\textit{Acknowledgement} \\ \text{cnt:}\textit{RecType} \end{bmatrix}$

Let us now check that we can characterize the types of information states of A and B in the dialogue (59), where we represent the information states associated with the two agents at various points in the dialogue as a_i and b_i, and the utterance events as u_i.

(59)

	a_0,b_0	
A:	Dudamel is a conductor	u_1
	a_1,b_1	
B:	Aha	u_2
	a_2,b_2	
A:	Beethoven is a composer	u_3
	a_3,b_3	
B:	ok	u_4
	a_4,b_4	
A:	Uchida is a pianist	u_5
	a_5,b_5	
B:	ok	u_6
	a_6,b_6	

We will assume that a_0 and b_0 are initial states, essentially empty except for A's agenda to make the three assertions. This is shown in (60).

(60)

a. a_0 :
$$\begin{bmatrix}
\text{private:} & \begin{bmatrix}
\text{agenda=[} & \begin{bmatrix} \text{e:}\textit{Assertion} \wedge_{\cdot} \left[\text{sp=}\textit{SELF}\text{:}\textit{Ind}\right] \\ \text{cnt=}\left[\text{e:conductor(dudamel)}\right]\text{:}\textit{RecType} \\ \text{c}_{\text{cnt}}\text{:content(e,cnt)} \end{bmatrix}, \\
 & \begin{bmatrix} \text{e:}\textit{Assertion} \wedge_{\cdot} \left[\text{sp=}\textit{SELF}\text{:}\textit{Ind}\right] \\ \text{cnt=}\left[\text{e:composer(beethoven)}\right]\text{:}\textit{RecType} \\ \text{c}_{\text{cnt}}\text{:content(e,cnt)} \end{bmatrix}, \\
 & \begin{bmatrix} \text{e:}\textit{Assertion} \wedge_{\cdot} \left[\text{sp=}\textit{SELF}\text{:}\textit{Ind}\right] \\ \text{cnt=}\left[\text{e:pianist(uchida)}\right]\text{:}\textit{RecType} \\ \text{c}_{\text{cnt}}\text{:content(e,cnt)} \end{bmatrix} \text{]:[}\textit{RecType}\text{]}
\end{bmatrix} \\
\text{shared:} & \begin{bmatrix} \text{latest-utterance:}\textit{Nil} \\ \text{commitments=[]:}\textit{RecType} \end{bmatrix}
\end{bmatrix}$$

b. b_0 :
$$\begin{bmatrix} \text{private:}\left[\text{agenda=[]:[}\textit{RecType}\text{]}\right] \\ \text{shared:}\begin{bmatrix} \text{latest-utterance:}\textit{Nil} \\ \text{commitments=[]:}\textit{RecType} \end{bmatrix} \end{bmatrix}$$

(60) indicates that a_0 is an appropriate argument to the function **ExecTopAgenda** given in (52) and repeated in Appendix C. The result of applying **ExecTopAgenda** to a_0 is given in (61).

(61) **ExecTopAgenda**(a_0) =
$$\begin{bmatrix}
\text{move:} & \begin{bmatrix} \text{e:}\textit{Assertion} \wedge_{\cdot} \left[\text{sp=}\textit{SELF}\text{:}\textit{Ind}\right] \\ \text{cnt=}\left[\text{e:conductor(dudamel)}\right]\text{:}\textit{RecType} \\ \text{c}_{\text{cnt}}\text{:content(e,cnt)} \end{bmatrix} \\
\text{chart:}\textit{Chart} & \\
\text{e:m-interp(chart,move)} &
\end{bmatrix}$$

Note that given our notational convention on using *SELF* (p. 46), (61) is actually a dependent type as in (62).

(62) λa:*Ind* .
$$\begin{bmatrix}
\text{move:} & \begin{bmatrix} \text{e:}\textit{Assertion}\wedge_{\cdot} \left[\text{sp=}a\text{:}\textit{Ind}\right] \\ \text{cnt=}\left[\text{e:conductor(dudamel)}\right]\text{:}\textit{RecType} \\ \text{c}_{\text{cnt}}\text{:content(e,cnt)} \end{bmatrix} \\
\text{chart:}\textit{Chart} & \\
\text{e:m-interp(chart,move)} &
\end{bmatrix}$$

This means that the appropriate licensing condition on type acts associated with **ExecTopAgenda** (given in Appendix C.1.2) is the *de se* variant in (63).

(63) If $f : (T \rightarrow (\textit{Ind} \rightarrow \textit{Type}))$ is an action function then for any object a and agent A, $a :_A T$ licenses $:_A f(a)(A)$!

That is, A is licensed to create (or contribute to the creation of) something of the type (61) with A itself as *SELF*. We have not yet said anything about what is involved in creating something of this type. The procedure involves generating a chart (in this chapter conceived of as a sign) whose content corresponds to the content of the move. We will not make this formally precise here but will wait until Chapter 3 where we have developed a more serious approach to grammar. Suffice it to say that the agent's resources must include the sign types introduced in (55) and that there is just one sign type here involving the type 'conductor(dudamel)' which figures in the content of the move specified in (61), namely sign("Dudamel is a conductor", conductor(dudamel)). For convenience we will abbreviate the notation of this type as $\Sigma_{\text{"Dudamel is a conductor"}}$. Only a sign of this type will satisfy m-interp for a move of the move type. Thus in order to realize something of type (61) A must in fact create something of type (64), a subtype of (61).

(64)
$$\left[\begin{array}{lcl} \text{move} & : & \left[\begin{array}{lcl} \text{e} & : & \textit{Assertion} \wedge\!\!\!\cdot\ \left[\text{sp=}\textit{SELF}\text{:}\textit{Ind}\right] \\ \text{cnt=}\left[\text{e:conductor(dudamel)}\right] & : & \textit{RecType} \\ \text{c}_{\text{cnt}} & : & \text{content(e,cnt)} \end{array}\right] \\ \text{chart} & : & \Sigma_{\text{"Dudamel is a conductor"}} \\ \text{e} & : & \text{m-interp(chart,move)} \end{array}\right]$$

(Here it is important that $\Sigma_{\text{"Dudamel is a conductor"}}$ is an abbreviation for the *notation* of the sign type, since when the notation is interpreted *in situ* in (64) each local path-name occurring as an argument to a predicate will be prefixed by 'chart.' and thus what occurs in the 'chart'-field of (64) is actually a modified version of the original type. It is possible to develop a notation that is more explicit but it becomes cluttered and unwieldy.)

Thus we can conclude that u_1 is judged by A to be of type (64). We can now predict that a_1 is of type **IntegrateOwnAssertion**$(a_0)(u_1)$ which, given the types we have hypothesized for a_0 and u_1 will be (65).

(65)
$$\left[\begin{array}{l}
\text{private:} \left[\begin{array}{l}
\text{agenda=[} \left[\begin{array}{l}
\text{e:}\textit{Acknowledgement} \dot{\wedge} \left[\begin{array}{l}\text{sp=}u_1\text{.move.e.au:}\textit{Ind}\\ \text{au=}\textit{SELF}\text{:}\textit{Ind}\end{array}\right]\\
\text{cnt=}u_1\text{.move.cnt:}\textit{RecType}\\
\text{c}_{\text{cnt}}\text{:content(e,cnt)}
\end{array}\right],\\
\left[\begin{array}{l}
\text{e:}\textit{Assertion} \dot{\wedge} \left[\text{sp=}\textit{SELF}\text{:}\textit{Ind}\right]\\
\text{cnt=}\left[\text{e:composer(beethoven)}\right]\text{:}\textit{RecType}\\
\text{c}_{\text{cnt}}\text{:content(e,cnt)}
\end{array}\right],\\
\left[\begin{array}{l}
\text{e:}\textit{Assertion} \dot{\wedge} \left[\text{sp=}\textit{SELF}\text{:}\textit{Ind}\right]\\
\text{cnt=}\left[\text{e:pianist(uchida)}\right]\text{:}\textit{RecType}\\
\text{c}_{\text{cnt}}\text{:content(e,cnt)}
\end{array}\right]\text{]:[}\textit{RecType}\text{]}
\end{array}\right]\\
\text{shared:} \left[\begin{array}{l}
\text{latest-utterance:} \left[\begin{array}{l}
\text{move=}u_1\text{.move:} \left[\begin{array}{l}
\text{e:}\textit{Assertion} \dot{\wedge} \left[\text{sp=}\textit{SELF}\text{:}\textit{Ind}\right]\\
\text{cnt=}\left[\text{e:conductor(dudamel)}\right]\text{:}\textit{RecType}\\
\text{c}_{\text{cnt}}\text{:content(e,cnt)}
\end{array}\right]\\
\text{chart=}u_1\text{.chart:}\Sigma_{\text{“Dudamel is a conductor”}}\\
\text{e=}u_1\text{.e:m-interp(chart,move)}
\end{array}\right]
\end{array}\right]
\end{array}\right]$$

We can now use (65) to update the type we had for a_0, given in (60a), as in (66a) which is identical with (66b).

(66)

a.
$$\begin{bmatrix}\text{private:}\begin{bmatrix}\text{agenda=[}\begin{bmatrix}\text{e:}\textit{Assertion}\wedge\!\!\cdot\ \begin{bmatrix}\text{sp=}\textit{SELF}\text{:}\textit{Ind}\end{bmatrix}\\ \text{cnt=}\begin{bmatrix}\text{e:conductor(dudamel)}\end{bmatrix}\text{:}\textit{RecType}\\ \text{c}_{\text{cnt}}\text{:content(e,cnt)}\end{bmatrix},\\ \begin{bmatrix}\text{e:}\textit{Assertion}\wedge\!\!\cdot\ \begin{bmatrix}\text{sp=}\textit{SELF}\text{:}\textit{Ind}\end{bmatrix}\\ \text{cnt=}\begin{bmatrix}\text{e:composer(beethoven)}\end{bmatrix}\text{:}\textit{RecType}\\ \text{c}_{\text{cnt}}\text{:content(e,cnt)}\end{bmatrix},\\ \begin{bmatrix}\text{e:}\textit{Assertion}\wedge\!\!\cdot\ \begin{bmatrix}\text{sp=}\textit{SELF}\text{:}\textit{Ind}\end{bmatrix}\\ \text{cnt=}\begin{bmatrix}\text{e:pianist(uchida)}\end{bmatrix}\text{:}\textit{RecType}\\ \text{c}_{\text{cnt}}\text{:content(e,cnt)}\end{bmatrix}\text{]:[}\textit{RecType}\text{]}\end{bmatrix}\\ \text{shared:}\begin{bmatrix}\text{latest-utterance:}\textit{Nil}\\ \text{commitments=[]:}\textit{RecType}\end{bmatrix}\end{bmatrix}$$

$\boxed{\wedge\!\!\cdot}$

$$\begin{bmatrix}\text{private:}\begin{bmatrix}\text{agenda=[}\begin{bmatrix}\text{e:}\textit{Acknowledgement}\wedge\!\!\cdot\ \begin{bmatrix}\text{sp=}u_1\text{.move.e.au:}\textit{Ind}\\ \text{au=}\textit{SELF}\text{:}\textit{Ind}\end{bmatrix}\\ \text{cnt=}u_1\text{.move.cnt:}\textit{RecType}\\ \text{c}_{\text{cnt}}\text{:content(e,cnt)}\end{bmatrix},\\ \begin{bmatrix}\text{e:}\textit{Assertion}\wedge\!\!\cdot\ \begin{bmatrix}\text{sp=}\textit{SELF}\text{:}\textit{Ind}\end{bmatrix}\\ \text{cnt=}\begin{bmatrix}\text{e:composer(beethoven)}\end{bmatrix}\text{:}\textit{RecType}\\ \text{c}_{\text{cnt}}\text{:content(e,cnt)}\end{bmatrix},\\ \begin{bmatrix}\text{e:}\textit{Assertion}\wedge\!\!\cdot\ \begin{bmatrix}\text{sp=}\textit{SELF}\text{:}\textit{Ind}\end{bmatrix}\\ \text{cnt=}\begin{bmatrix}\text{e:pianist(uchida)}\end{bmatrix}\text{:}\textit{RecType}\\ \text{c}_{\text{cnt}}\text{:content(e,cnt)}\end{bmatrix}\text{]:[}\textit{RecType}\text{]}\end{bmatrix}\\ \text{shared:}\begin{bmatrix}\text{latest-utterance:}\begin{bmatrix}\text{move=}u_1\text{.move:}\begin{bmatrix}\text{e:}\textit{Assertion}\wedge\!\!\cdot\ \begin{bmatrix}\text{sp=}\textit{SELF}\text{:}\textit{Ind}\end{bmatrix}\\ \text{cnt=}\begin{bmatrix}\text{e:conductor(dudamel)}\end{bmatrix}\text{:}\textit{RecType}\\ \text{c}_{\text{cnt}}\text{:content(e,cnt)}\end{bmatrix}\\ \text{chart=}u_1\text{.chart:}\Sigma_{\text{“Dudamel is a conductor”}}\\ \text{e=}u_1\text{.e:m-interp(chart,move)}\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

b.
$$\begin{bmatrix}\text{private:}\begin{bmatrix}\text{agenda=[}\begin{bmatrix}\text{e:}\textit{Acknowledgement}\wedge\!\!\cdot\ \begin{bmatrix}\text{sp=}u_1\text{.move.e.au:}\textit{Ind}\\ \text{au=}\textit{SELF}\text{:}\textit{Ind}\end{bmatrix}\\ \text{cnt=}u_1\text{.move.cnt:}\textit{RecType}\\ \text{c}_{\text{cnt}}\text{:content(e,cnt)}\end{bmatrix},\\ \begin{bmatrix}\text{e:}\textit{Assertion}\wedge\!\!\cdot\ \begin{bmatrix}\text{sp=}\textit{SELF}\text{:}\textit{Ind}\end{bmatrix}\\ \text{cnt=}\begin{bmatrix}\text{e:composer(beethoven)}\end{bmatrix}\text{:}\textit{RecType}\\ \text{c}_{\text{cnt}}\text{:content(e,cnt)}\end{bmatrix},\\ \begin{bmatrix}\text{e:}\textit{Assertion}\wedge\!\!\cdot\ \begin{bmatrix}\text{sp=}\textit{SELF}\text{:}\textit{Ind}\end{bmatrix}\\ \text{cnt=}\begin{bmatrix}\text{e:pianist(uchida)}\end{bmatrix}\text{:}\textit{RecType}\\ \text{c}_{\text{cnt}}\text{:content(e,cnt)}\end{bmatrix}\text{]:[}\textit{RecType}\text{]}\end{bmatrix}\\ \text{shared:}\begin{bmatrix}\text{latest-utterance:}\begin{bmatrix}\text{move=}u_1\text{.move:}\begin{bmatrix}\text{e:}\textit{Assertion}\wedge\!\!\cdot\ \begin{bmatrix}\text{sp=}\textit{SELF}\text{:}\textit{Ind}\end{bmatrix}\\ \text{cnt=}\begin{bmatrix}\text{e:conductor(dudamel)}\end{bmatrix}\text{:}\textit{RecType}\\ \text{c}_{\text{cnt}}\text{:content(e,cnt)}\end{bmatrix}\\ \text{chart=}u_1\text{.chart:}\Sigma_{\text{“Dudamel is a conductor”}}\\ \text{e=}u_1\text{.e:m-interp(chart,move)}\end{bmatrix}\\ \text{commitments=[]:}\textit{RecType}\end{bmatrix}\end{bmatrix}$$

Thus we can conclude that a_1 is of type (66b). A type for b_1 can be obtained in a similar fashion using **IntegrateOtherAssertion**$(b_0)(u_1)$. The type that B will assign to u_1 can be predicted by the perception function (Appendix C) in (67).

(67) $\lambda e{:}\begin{bmatrix}\text{e:“Dudamel is a conductor”}\\ \text{au=}\textit{SELF}\text{:}\textit{Ind}\end{bmatrix}$.
$$\begin{bmatrix} \text{move} & : & \begin{bmatrix} \text{e} & : & \textit{SpeechAct} \wedge\!\!\!\cdot\ \left[\text{au=}\textit{SELF}\text{:}\textit{Ind}\right] \\ \text{cnt} & : & \textit{Cnt} \\ \text{c}_{\text{cnt}} & : & \text{content(e,cnt)} \end{bmatrix} \\ \text{chart} & : & \Sigma_{\text{“Dudamel is a conductor”}} \\ \text{e} & : & \text{m-interp(chart,move)} \end{bmatrix}$$

(67) together with the type we have for b_0 predicts that **IntegrateOtherAssertion**$(b_0)(u_1)$ will be the type (68).

(68)
$$\begin{bmatrix} \text{private:} \begin{bmatrix} \text{agenda=} \ [\begin{bmatrix} \text{e:}\textit{Acknowledgement} \wedge\!\!\!\cdot\ \left[\text{sp=}\textit{SELF}\text{:}\textit{Ind}\right] \\ \text{cnt=}\left[\text{e:conductor(dudamel)}\right]\text{:}\textit{RecType} \\ \text{c}_{\text{cnt}}\text{:content(e,cnt)} \end{bmatrix} \ ] \ \text{:[}\textit{RecType}\text{]} \end{bmatrix} \\ \text{shared:} \begin{bmatrix} \text{latest-utterance:} \begin{bmatrix} \text{move=}u_1\text{.move:} \begin{bmatrix} \text{e:}\textit{Assertion} \wedge\!\!\!\cdot\ \left[\text{au=}\textit{SELF}\text{:}\textit{Ind}\right] \\ \text{cnt=}\left[\text{e:conductor(dudamel)}\right]\text{:}\textit{RecType} \\ \text{c}_{\text{cnt}}\text{:content(e,cnt)} \end{bmatrix} \\ \text{chart=}u_1\text{.chart: } \Sigma_{\text{“Dudamel is a conductor”}} \\ \text{e=}u_1\text{.e:m-interp(chart,move)} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

We can now use (68) to update the type we had for b_0, obtaining (69a) identical with (69b).

(69)

a.
$$\begin{bmatrix}\text{private:}\begin{bmatrix}\text{agenda=[]:}[\textit{RecType}]\end{bmatrix}\\ \text{shared:}\begin{bmatrix}\text{latest-utterance:}\textit{Nil}\\ \text{commitments=[]:}\textit{RecType}\end{bmatrix}\end{bmatrix}\ \boxed{\wedge\!\!\!\cdot}$$
$$\begin{bmatrix}\text{private:}\begin{bmatrix}\text{agenda= [}\begin{bmatrix}\text{e:}\textit{Acknowledgement}\wedge\!\!\!\cdot\ \begin{bmatrix}\text{sp=}\textit{SELF}\text{:}\textit{Ind}\end{bmatrix}\\ \text{cnt=}\begin{bmatrix}\text{e:conductor(dudamel)}\end{bmatrix}\text{:}\textit{RecType}\\ \text{c}_{\text{cnt}}\text{:content(e,cnt)}\end{bmatrix}\text{] :[}\textit{RecType}\text{]}\end{bmatrix}\\ \text{shared:}\begin{bmatrix}\text{latest-utterance:}\begin{bmatrix}\text{move=}u_1\text{.move:}\begin{bmatrix}\text{e:}\textit{Assertion}\wedge\!\!\!\cdot\ \begin{bmatrix}\text{au=}\textit{SELF}\text{:}\textit{Ind}\end{bmatrix}\\ \text{cnt=}\begin{bmatrix}\text{e:conductor(dudamel)}\end{bmatrix}\text{:}\textit{RecType}\\ \text{c}_{\text{cnt}}\text{:content(e,cnt)}\end{bmatrix}\\ \text{chart=}u_1\text{.chart: }\Sigma_{\text{“Dudamel is a conductor”}}\\ \text{e=}u_1\text{.e:m-interp(chart,move)}\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

b.
$$\begin{bmatrix}\text{private:}\begin{bmatrix}\text{agenda= [}\begin{bmatrix}\text{e:}\textit{Acknowledgement}\wedge\!\!\!\cdot\ \begin{bmatrix}\text{sp=}\textit{SELF}\text{:}\textit{Ind}\end{bmatrix}\\ \text{cnt=}\begin{bmatrix}\text{e:conductor(dudamel)}\end{bmatrix}\text{:}\textit{RecType}\\ \text{c}_{\text{cnt}}\text{:content(e,cnt)}\end{bmatrix}\text{] :[}\textit{RecType}\text{]}\end{bmatrix}\\ \text{shared:}\begin{bmatrix}\text{latest-utterance:}\begin{bmatrix}\text{move=}u_1\text{.move:}\begin{bmatrix}\text{e:}\textit{Assertion}\wedge\!\!\!\cdot\ \begin{bmatrix}\text{au=}\textit{SELF}\text{:}\textit{Ind}\end{bmatrix}\\ \text{cnt=}\begin{bmatrix}\text{e:conductor(dudamel)}\end{bmatrix}\text{:}\textit{RecType}\\ \text{c}_{\text{cnt}}\text{:content(e,cnt)}\end{bmatrix}\\ \text{chart=}u_1\text{.chart: }\Sigma_{\text{“Dudamel is a conductor”}}\\ \text{e=}u_1\text{.e:m-interp(chart,move)}\end{bmatrix}\end{bmatrix}\\ \text{commitments=[]:}\textit{RecType}\end{bmatrix}$$

Now we are in a situation where both A and B are in information states (a_1 and b_1) with non-empty agendas. But they are coordinated in that A has an acknowledgement to be spoken by B topmost on the agenda and B has an acknowledgement with the same content to be spoken by B with A as the audience. **ExecTopAgenda** is applicable to both a_1 and b_1. (70a) is **ExecTopAgenda**(a_1) and (70b) is **ExecTopAgenda**(b_1).

(70)

a.
$$\begin{bmatrix}\text{move} & : & \begin{bmatrix}\text{e:}\textit{Acknowledgement}\wedge\!\!\!\cdot\ \begin{bmatrix}\text{sp=}u_1\text{.move.e.au:}\textit{Ind}\\ \text{au=}\textit{SELF}\text{:}\textit{Ind}\end{bmatrix}\\ \text{cnt=}u_1\text{.move.cnt:}\textit{RecType}\\ \text{c}_{\text{cnt}}\text{:content(e,cnt)}\end{bmatrix}\\ \text{chart} & : & \textit{Chart}\\ \text{e} & : & \text{m-interp(chart,move)}\end{bmatrix}$$

b.
$$\begin{bmatrix}\text{move} & : & \begin{bmatrix}\text{e:}\textit{Acknowledgement}\wedge\!\!\!\cdot\ \begin{bmatrix}\text{sp=}\textit{SELF}\text{:}\textit{Ind}\\ \text{au=}u_1\text{.move.e.sp:}\textit{Ind}\end{bmatrix}\\ \text{cnt=}\begin{bmatrix}\text{e:conductor(dudamel)}\end{bmatrix}\text{:}\textit{RecType}\\ \text{c}_{\text{cnt}}\text{:content(e,cnt)}\end{bmatrix}\\ \text{chart} & : & \textit{Chart}\\ \text{e} & : & \text{m-interp(chart,move)}\end{bmatrix}$$

By substituting values for *SELF* and values from u_1 we can see the extent to which A and B are coordinated. Both (70a) and (70b) reduce to the type in (71).

(71)
$$\begin{bmatrix} \text{move} & : & \begin{bmatrix} \text{e:}\textit{Acknowledgement} \wedge\!\!\!\cdot \begin{bmatrix}\text{sp=}B\text{:}\textit{Ind}\\ \text{au=}A\text{:}\textit{Ind}\end{bmatrix} \\ \text{cnt=}\left[\text{e:conductor(dudamel)}\right]\text{:}\textit{RecType} \\ \text{c}_{\text{cnt}}\text{:content(e,cnt)}\end{bmatrix} \\ \text{chart} & : & \textit{Chart} \\ \text{e} & : & \text{m-interp(chart,move)}\end{bmatrix}$$

Both A and B can now, in virtue of **ExecTopAgenda** play their respective roles in creating an event of type (71), A by waiting for and paying attention to the acknowledgement and B by uttering the acknowledgement. This is an elementary form of what is known as *turn-taking* in the dialogue literature (Sacks *et al.*, 1974). The acknowledgement is u_2 in (59). In virtue of what is on the top of their respective agendas both A and B can judge u_2 to be of the type (71). A and B can now update their gameboards in virtue of **IntegrateOtherAcknowledgement** and **IntegrateOwnAcknowledgement** respectively. A will thus judge a_2 to be of type (72a) as a result of updating (66b) and B will judge b_2 to be of type (72b) as a result of updating (69b).

(72) a.
$$\begin{bmatrix} \text{private:} & \begin{bmatrix} \text{agenda=[} \begin{bmatrix}\text{e:}\textit{Assertion} \wedge\!\!\!\cdot \left[\text{sp=}\textit{SELF}\text{:}\textit{Ind}\right] \\ \text{cnt=}\left[\text{e:composer(beethoven)}\right]\text{:}\textit{RecType} \\ \text{c}_{\text{cnt}}\text{:content(e,cnt)}\end{bmatrix}, \\ \begin{bmatrix}\text{e:}\textit{Assertion} \wedge\!\!\!\cdot \left[\text{sp=}\textit{SELF}\text{:}\textit{Ind}\right] \\ \text{cnt=}\left[\text{e:pianist(uchida)}\right]\text{:}\textit{RecType} \\ \text{c}_{\text{cnt}}\text{:content(e,cnt)}\end{bmatrix} \text{]:[}\textit{RecType}\text{]}\end{bmatrix} \\ \text{shared:} & \begin{bmatrix} \text{latest-utterance:} & \begin{bmatrix} \text{move=}u_2\text{.move:} \begin{bmatrix}\text{e:}\textit{Acknowledgement} \wedge\!\!\!\cdot \left[\text{au=}\textit{SELF}\text{:}\textit{Ind}\right] \\ \text{cnt=}\left[\text{e:conductor(dudamel)}\right]\text{:}\textit{RecType} \\ \text{c}_{\text{cnt}}\text{:content(e,cnt)}\end{bmatrix} \\ \text{chart=}u_2\text{.chart:}\Sigma_{\text{“Aha”}} \\ \text{e=}u_2\text{.e:m-interp(chart,move)}\end{bmatrix} \\ \text{commitments=}\begin{bmatrix}\text{prev=[]:}\textit{RecType} \\ \text{e:conductor(dudamel)}\end{bmatrix}\text{:}\textit{RecType}\end{bmatrix}\end{bmatrix}$$

b.
$$\begin{bmatrix} \text{private:}\left[\text{agenda= [] :[}\textit{RecType}\text{]}\right] \\ \text{shared:} & \begin{bmatrix} \text{latest-utterance:} & \begin{bmatrix} \text{move=}u_2\text{.move:} \begin{bmatrix}\text{e:}\textit{Acknowledgement} \wedge\!\!\!\cdot \left[\text{sp=}\textit{SELF}\text{:}\textit{Ind}\right] \\ \text{cnt=}\left[\text{e:conductor(dudamel)}\right]\text{:}\textit{RecType} \\ \text{c}_{\text{cnt}}\text{:content(e,cnt)}\end{bmatrix} \\ \text{chart=}u_2\text{.chart: }\Sigma_{\text{“Aha”}} \\ \text{e=}u_2\text{.e:m-interp(chart,move)}\end{bmatrix} \\ \text{commitments=}\begin{bmatrix}\text{prev=[]:}\textit{RecType} \\ \text{e:conductor(dudamel)}\end{bmatrix}\text{:}\textit{RecType}\end{bmatrix}\end{bmatrix}$$

A and B are coordinated in that they both hypothesize the same type for shared.commitments. Now the assertion-acknowledgement cycle can begin again and repeat until both agents have gameboards with empty agendas. The final gameboards for A and B respectively are given in (73).

(73)

a.
$$\begin{bmatrix}\text{private:}\begin{bmatrix}\text{agenda=[]:[}\textit{RecType}\text{]}\end{bmatrix}\\ \text{shared:}\begin{bmatrix}\text{latest-utterance:}\begin{bmatrix}\text{move=}u_6\text{.move:}\begin{bmatrix}\text{e:}\textit{Acknowledgement} \wedge_{\cdot} \begin{bmatrix}\text{au=}\textit{SELF}\text{:}\textit{Ind}\end{bmatrix}\\ \text{cnt=}\begin{bmatrix}\text{e:pianist(uchida)}\end{bmatrix}\text{:}\textit{RecType}\\ \text{c}_{\text{cnt}}\text{:content(e,cnt)}\end{bmatrix}\\ \text{chart=}u_6\text{.chart:}\Sigma_{\text{“ok”}}\\ \text{e=}u_6\text{.e:m-interp(chart,move)}\end{bmatrix}\\ \text{commitments=}\begin{bmatrix}\text{prev:}\begin{bmatrix}\text{prev:}\begin{bmatrix}\text{prev=[]:}\textit{RecType}\\ \text{e:conductor(dudamel)}\end{bmatrix}\\ \text{e:composer(beethoven)}\end{bmatrix}\\ \text{e:pianist(uchida)}\end{bmatrix}\text{:}\textit{RecType}\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

b.
$$\begin{bmatrix}\text{private:}\begin{bmatrix}\text{agenda=[] :[}\textit{RecType}\text{]}\end{bmatrix}\\ \text{shared:}\begin{bmatrix}\text{latest-utterance:}\begin{bmatrix}\text{move=}u_6\text{.move:}\begin{bmatrix}\text{e:}\textit{Acknowledgement} \wedge_{\cdot} \begin{bmatrix}\text{sp=}\textit{SELF}\text{:}\textit{Ind}\end{bmatrix}\\ \text{cnt=}\begin{bmatrix}\text{e:pianist(uchida)}\end{bmatrix}\text{:}\textit{RecType}\\ \text{c}_{\text{cnt}}\text{:content(e,cnt)}\end{bmatrix}\\ \text{chart=}u_6\text{.chart: }\Sigma_{\text{“ok”}}\\ \text{e=}u_6\text{.e:m-interp(chart,move)}\end{bmatrix}\\ \text{commitment=}\begin{bmatrix}\text{prev:}\begin{bmatrix}\text{prev:}\begin{bmatrix}\text{prev=[]:}\textit{RecType}\\ \text{e:conductor(dudamel)}\end{bmatrix}\\ \text{e:composer(beethoven)}\end{bmatrix}\\ \text{e:pianist(uchida)}\end{bmatrix}\text{:}\textit{RecType}\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

Chapter 3

Grammar

In Chapter 2 we made the simplifying assumption that sentences come as single unanalyzed units (something like the assumption that is made in propositional logic). In this chapter we will deal with the same simple examples but break the sentences down into their constituent parts. (This will be something like moving from propositional logic to predicate logic without quantifiers.) In order to do this we will need more complex signs.

We will first consider how linguistic constituent structure is related to our general perception of events. We have so far talked of events in terms of string types which we have related to finite state automata. Finite state automata are equivalent to regular grammars. We will now consider an example of how we perceive events which suggest a more complex structure in terms of strings of regular types. This gives us something which is equivalent to recursive transition networks (RTNs) which are in turn equivalent to context free grammars.[1] Consider an event type of bus trips, *BusTrip*. This could be defined as in (1).

(1) $BusTrip \equiv GetBus^\frown TravelOnBus^\frown GetOffBus$

Each of the three event types which are concatenated in (1) could be further broken down into strings of events. For example, *GetBus* might be defined as in (2).

(2) $GetBus \equiv WaitAtBusstop^{*\frown} BusArrive^\frown GetOnBus$

The elements in (2) could be broken down further. For example, getting on the bus could be

[1] For a general introduction to automata theory and its relation to the Chomsky hierarchy see, for example, Partee *et al.* (1990).

analyzed in terms of going towards a door on the bus, waiting for the door to open, placing one foot on the step into the bus and then the other, paying for your ticket and so on. There seems almost no limit to how finegrained an analysis of events we can give. Which muscles do you have to move in order to place your right foot inside the bus? What events are involved in the contraction of this muscle? However, there seems to be a limit on the level of detail we need to be conscious of (or even are capable of being conscious of) in order to carry out a high level action like getting on a bus. We can also build upwards from the type *BusTrip*. For example, many bus trips are not direct in that we have to change buses in order to reach our destination. Thus a bus trip can consist of a string of events where you get on a bus, travel on it and then get off it again. A return bus trip involves a bus trip from one place to another followed (after intervening events) by a bus trip from the second place back to the first. Both of those bus trips might involve several buses if the connection is not direct.

The notation we have used in (1) and (2) is used to mean that what occurs to the left of $\equiv$ is a convenient notational abbreviation for what occurs on the right. That is, whenever we write the symbol on the left, that is just shorthand for the longer expression on the right. Given the two definitions in (1) and (2), *BusTrip* is just an abbreviation for the regular string type in (3).

(3) *WaitAtBusstop**⌢*BusArrive*⌢*GetOnBus*⌢*TravelOnBus*⌢*GetOffBus*

Thus while our notation is giving us the beginnings of a hierarchical organization, the type that is represented by the notation is not hierarchically organized. We are still in the realm of a finite state system. Compare this with the statements in (4).

(4) a. e : *BusTrip* iff e : *GetBus*⌢*TravelOnBus*⌢*GetOffBus*

b. e : *GetBus* iff e : *WaitAtBusstop**⌢*BusArrive*⌢*GetOnBus*

The statements in (4) claim that there are distinct types *BusTrip* and *GetBus* in addition to the regular types used on the right-hand side of 'iff'. These types are *equivalent* to the regular types in the sense that anything of the one type will be of the other type. Now the actual type system (not just the notation) is hierarchically organized and includes two additional "higher" types *BusTrip* and *GetBus*. On the face of it one might think that the type system with the additional higher types would be just a more complicated way of achieving the same result and would be less efficient than a system which just includes the regular types. However, there seems to be good reason to suppose that an organism that organizes its event perception in terms of such a hierarchical type system would have serious advantages over an organism that lacks the hierachical organization. These advantages include at least the following:

access and compact representation Recall from Chapter 1 that we want to consider the types that an agent has available as resources as being represented in the brain states of the agent.

Having higher types means that something corresponding to a complex type can be stored as a single element. In a complex reasoning task this can give considerable advantage in that the task can be represented in a more compact fashion and it can be easier to access (search and find) something which is a single element rather than something which is represented in terms of a complex string each element of which has to be checked in order to be sure that you have found the right element.

planning Having a compact representation facilitates planning. It is feasible to plan to take a bus trip given that we can conceive of it as such without having to plan for all the small subevents that make it up, for example, all that is involved in lifting your legs in the right way in order get on the bus. The ability to plan actions seems based on an ability to classify events in a hierarchical way.

reuse A hierarchical organization of event types means that certain event types can be reused in other event types. For example, getting on a bus (waiting for the doors to open, putting one foot inside and so on) can be very much like getting on a train. Similarly, paying for a ticket on a bus trip involves an exchange of money for a ticket in much the same way for a bus, a tram, a train, a theatre performance and so on. An agent which is not able to perceive this kind of generalization would at best use up a lot of memory coding the same event types over and over as parts of different larger event types.

learning The hierarchical organization of event types and the reuse capabilities it offers also facilitates learning of new event types. In learning to take the tram it can be useful to reuse what you have learnt about buying tickets on buses and insert it ready made into your type for tram trips. If it turns out that the procedure for buying tickets for trams is slightly different from for buses (for example, you can buy a ticket on the bus but you have to pay before you get on the tram) you nevertheless have a buying ticket type which you can modify. This might involve creating more types corresponding to those strings which the two ticket buying procedures have in common to separate out the differences between the two procedures.

Related observations about the importance of hierarchical structure for behaviour and its relationship to hierarchical reinforcement learning and neurological structure have been made for example by Botvinick (2008); Botvinick *et al.* (2009); Ribas-Fernandes *et al.* (2011).

Introducing hierarchical types in this way is an important step in our cognitive processing of events because of the computational and learning processes indicated above even if the class of events we are formally able to recognize is the same as what could be recognized by non-hierarchical regular string types, that is, technically, finite state languages. An organism with hierarchically organized types will have important advantages in acquiring new finite state event patterns. An evolutionary step from non-hierarchically organized string types to hierarchically organized types is a significant development and organisms with hierarchical types will have clear evolutionary advantages over those that do not.

However, hierarchical organization brings with it, almost as a kind of side effect, something which means that the organism could recognize classes of events that are not finite state. This is known as *recursion*. Hierarchical organization means that we can give type definitions of the form in (5).

(5) $a : T$ iff $a : T_1^\frown \ldots ^\frown T_n$

If we do not explicitly rule it out there is nothing to say that one of the T_i is not T itself. Of course, things will go badly wrong if we have a definition such as (6).

(6) $a : T$ iff $a : T_1^\frown T^\frown T_2$

If we try to perceive or create something of this type we will not be able to terminate and get into an endless string of objects of type T_1 and never be able to move on to T_2. However, if we define T in terms of a join type where at least one of the types in the join does not contain T, things will work fine. For example, (7):

(7) $a : T$ iff $a : (T_1^\frown T^\frown T_2 \vee T_1^\frown T_2)$

According to (7) anything of type T will be a string of objects of type T_1 followed by a string of equal length of objects of type T_2. It is the requirement "of equal length" which means that this type is not a regular type. For example, we could have the the regular type $T_1^{+\frown} T_2^+$ but this only expresses that we require a non-empty string of objects of type T_1 followed by a non-empty string of objects of type T_2 without the equal length requirement. What we have done here is restate a basic result from formal language theory in terms of our types. In formal language theory one talks of languages of the form $a^n b^m$ (the set of strings of n a's followed by a string of m b's, for any n and m greater than 0) which is a regular or finite state language and $a^n b^n$ (the set of strings of n a's followed by n b's, for any n greater than 0) which is context free. While this possibility of recursion is offered as soon as we allow the hierarchical typing of events in this way, it is not clear that it is exploited to a great extent in non-linguistic events. The clear examples that seem to exist are examples like opening and closing Chinese boxes, that is, boxes within boxes. The type of opening and closing (reassembling) a Chinese box could be characterized as the $a^n b^n$-type in (8).

(8) e : *OpenClose* iff
e : (*Open*$^\frown$ *OpenClose*$^\frown$ *Close* $\vee$ *Open*$^\frown$ *Close*)

It is significant in this kind of example that the ordering of the events is forced on the agent by the physical reality of the boxes. There is only one order in which you can open all the boxes and only one order (the reverse order) in which you can close them if you are going to assemble all the boxes within a single box. It is unclear that such ordering is required in non-linguistic event types when it is not dictated by physical reality.

3.1 Syntax

We now turn our attention to how this hierarchical organization is reflected in the nature of linguistic events. In Chapter 2 we used (9) as our sign type.

(9) $$\begin{bmatrix} \text{s-event} & : & \mathit{SEvent} \\ \text{cnt} & : & \mathit{Cnt} \end{bmatrix}$$

This represents the pairing of a speech event with content in a Saussurean sign. It does not, however, require the presence of any hierarchical information in the sign corresponding to what in linguistic theory is normally referred to as the *constituent* (or *phrase*) structure of the utterance. To some extent it is arbitrary where we add this information. We could, for example, add it under the label 's-event', perhaps by dividing 's-event.e' into two fields 'phon' and 'syn' ("syntax"). However, it will be more convenient (in terms of keeping paths that we need to refer to often shorter) to add a third field labelled 'syn' at the top level of the sign type as in (10).

(10) $$\begin{bmatrix} \text{s-event} & : & \mathit{SEvent} \\ \text{syn} & : & \mathit{Syn} \\ \text{cnt} & : & \mathit{Cnt} \end{bmatrix}$$

However, as we will see below, *Syn* will require a 'daughters'-field for a string of signs. This means that *Sign* becomes a recursive type. It will be a *basic* type with its witnesses defined by (11).

(11) $\sigma : \mathit{Sign}$ iff $\sigma : \begin{bmatrix} \text{s-event} & : & \mathit{SEvent} \\ \text{syn} & : & \mathit{Syn} \\ \text{cnt} & : & \mathit{Cnt} \end{bmatrix}$

We shall take *Syn* to be the type (12).[2]

[2]One might think that *Syn* should also be defined as a recursive type since it can contain *Sign* which in its turn

(12)
$$\begin{bmatrix} \text{cat} & : & \mathit{Cat} \\ \text{daughters} & : & \mathit{Sign}^* \end{bmatrix}$$

The type *Sign*, as so far defined, can be seen as a *universal resource*. By this we mean that it is a type which is available for all languages. *Cat* is the type of names of syntactic categories. In this chapter we will take the witnesses of *Cat* to be: s ("sentence"), np ("noun phrase"), det ("determiner"), n ("noun"), v ("verb") and vp ("verb phrase"). These correspond to the categories we will use to cover the expressions of the fragment of English we introduced in Chapter 2. We will use capitalized versions of these category names to represent types of signs with the appropriate path in a sign type as in (13).

(13) a. $S \equiv \mathit{Sign} \wedge \left[\text{syn:}\left[\text{cat=s:}\mathit{Cat}\right]\right]$

b. $NP \equiv \mathit{Sign} \wedge \left[\text{syn:}\left[\text{cat=np:}\mathit{Cat}\right]\right]$

c. $\mathit{Det} \equiv \mathit{Sign} \wedge \left[\text{syn:}\left[\text{cat=det:}\mathit{Cat}\right]\right]$

d. $N \equiv \mathit{Sign} \wedge \left[\text{syn:}\left[\text{cat=n:}\mathit{Cat}\right]\right]$

e. $V \equiv \mathit{Sign} \wedge \left[\text{syn:}\left[\text{cat=v:}\mathit{Cat}\right]\right]$

f. $VP \equiv \mathit{Sign} \wedge \left[\text{syn:}\left[\text{cat=vp:}\mathit{Cat}\right]\right]$

Recall that the symbol $\wedge$ represents the merge operation on types as defined in Appendix A.12. This means that, for example, (13a) is the type in (14).

(14)
$$\begin{bmatrix} \text{s-event} & : & \begin{bmatrix} \text{e-loc} & : & \mathit{Loc} \\ \text{sp} & : & \mathit{Ind} \\ \text{au} & : & \mathit{Ind} \\ \text{e} & : & \mathit{Phon} \\ \text{c}_{\text{loc}} & : & \text{loc(e,e-loc)} \\ \text{c}_{\text{sp}} & : & \text{speaker(e,sp)} \\ \text{c}_{\text{au}} & : & \text{audience(e,au)} \end{bmatrix} \\ \text{syn} & : & \begin{bmatrix} \text{cat=s} & : & \mathit{Cat} \\ \text{daughters} & : & \mathit{Sign}^* \end{bmatrix} \\ \text{cnt} & : & \mathit{Cnt} \end{bmatrix}$$

can contain *Syn*. However, in the types we are currently proposing the only way for *Syn* to recur is through *Sign* and it is sufficient for *Sign* to be defined recursively to ensure that we do not introduce record types that are non-well founded sets of ordered pairs. That is, we want to avoid the mathematical object which is the type being a set which contains itself. In contrast the set of witnesses for a recursive type, while it will be infinite, will be well-founded.

We might think that the type *Cat* is a language specific resource and indeed if we were being more precise we might introduce separate types for different languages such as Cat_{eng}, Cat_{swe} and Cat_{tag} for the type of category names of English, Swedish and Tagalog respectively. However, there is a strong intuition that categories in different languages are more or less related. For example, we would not be surprised to find that the categories available for English and Swedish closely overlap (despite the fact that their internal syntactic structure differs) whereas the categories of English and Tagalog have less overlap. (See Gil, 2000 for discussion.) For this reason we assume that there is a universal resource *Cat* and that each language will have a subtype of *Cat* which specifies which of the categories are used in that particular language. This is related to the kind of view of linguistic universals as a kind of toolbox from which languages can choose which is put forward by Jackendoff (2002).

The ontological status of objects of type *Cat* as we have presented them is a little suspicious. Intuitively, categories should be subtypes of *Sign*, that is, like the types such as *S*, *NP* and so on in (13). We have identified signs belonging to these types as containing a particular object in *Cat* in their 'cat'-field. But one might try to characterize such signs in a different way, for example,as fulfilling certain conditions such as having certain kinds of daughters. However, this is not quite enough, for example, for lexical categories, which do not have daughters. We have to have a way of assigning categories to words and we need to create something in the sign-type that will indicate the arbitrary assignment of a category to a word. For want of a better solution we will introduce the category names which belong to the type *Cat* as a kind of "book-keeping" device that will identify a sign-type as being one whose witnesses belong to category bearing that name.

The 'daughters'-field is required to be a string of signs, possibly the empty string, since the type Sign^* uses the Kleene-*, that is the type of strings of signs including the empty string, ε. (See Appendix A.13.) Lexical items, that is words and phrases which are entered in the lexicon, will be related to signs which have the empty string of daughters. We will use *NoDaughters* to represent the type $\left[\text{syn:}\left[\text{daughters=}\varepsilon\text{:}\mathit{Sign}^*\right]\right]$.

If T_{phon} is a type (normally a phonological type, that is, $T_{\text{phon}} \sqsubseteq \mathit{Phon}$) and T_{sign} is a type (normally a sign type, that is,$T_{\text{sign}} \sqsubseteq \mathit{Sign}$, then we shall use Lex(T_{phon}, T_{sign}) to represent (15)

(15) $\quad T_{\text{sign}} \mathbin{\dot{\wedge}} \left[\text{s-event:}\left[\text{e:}T_{\text{phon}}\right]\right] \mathbin{\dot{\wedge}} \mathit{NoDaughters}$

This means, for example, that (16a) represents the type in (16b) which, after spelling out the abbreviations, can be seen to be the type in (16c).

(16) a. Lex("Dudamel", *NP*)

b. $NP \wedge\!\!\!\cdot\ \left[\text{s-event:}\left[\text{e:"Dudamel"}\right]\right] \wedge\!\!\!\cdot\ \textit{NoDaughters}$

c.
$$\left[\begin{array}{lcl}
\text{s-event} & : & \left[\begin{array}{lcl}
\text{e-loc} & : & \textit{Loc} \\
\text{sp} & : & \textit{Ind} \\
\text{au} & : & \textit{Ind} \\
\text{e} & : & \text{"Dudamel"} \\
\text{c}_{\text{loc}} & : & \text{loc(e,e-loc)} \\
\text{c}_{\text{sp}} & : & \text{speaker(e,sp)} \\
\text{c}_{\text{au}} & : & \text{audience(e,au)}
\end{array}\right] \\
\text{syn} & : & \left[\begin{array}{lcl}
\text{cat=np} & : & \textit{Cat} \\
\text{daughters=}\varepsilon & : & \textit{Sign}^*
\end{array}\right] \\
\text{cnt} & : & \textit{Cnt}
\end{array}\right]$$

We can think of 'Lex' as the function in (17)[3]

(17) λT_1:*Type*
λT_2:*Type* .
$T_1 \wedge\!\!\!\cdot\ \left[\text{s-event:}\left[\text{e:}T_2\right]\right] \wedge\!\!\!\cdot\ \textit{NoDaughters}$

This function, which creates sign types for lexical items in a language, associating types with a syntactic category, can be seen as a universal resource. We can think of it as representing a (somewhat uninteresting, but nevertheless true) linguistic universal: "There can be speech events of given types which have no daughters (lexical items)".

The lexical resources needed to cover our example fragment is given in (18).

(18) Lex("Dudamel", *NP*)
Lex("Beethoven", *NP*)
Lex("a", *Det*)
Lex("composer", *N*)
Lex("conductor", *N*)
Lex("is", *V*)
Lex("ok", *S*)
Lex("aha",*S*)

The types in (18) belong to the specific resources required for English. This is not to say that these

[3]We are using the notational convention for function application as used, for example, by Montague (1973) that if f is a function $f(a,b)$ is $f(b)(a)$.

resources cannot be shared with other languages. Proper names like *Dudamel* and *Beethoven* have a special status in that they can be reused in any language, though often in modified form, at least in terms of the phonological type with which they are associated without this being perceived as quotation, code-switching or simply showing off that you know another language.

Resources like (18) can be exploited by update rules. If Lex(T_w, C) is one of the lexical resources available to an agent A and A judges an event e to be of type T_w, then A is licensed to update their gameboard with the type Lex(T_w, C). Intuitively, this means that if the agent hears an utterance of the word "composer", then they can conclude that they have heard a sign which has the category noun. This is the beginning of *parsing*, which we will regard as the same kind of update involved in event perception as discussed in the previous chapters. The licensing condition corresponding to lexical resources like (18) is given in (19). We will return below to how this relates to gameboard update.

(19) If Lex(T, C) is a resource available to agent A, then for any u, $u :_A T$ licenses $:_A$ Lex(T, C) $\wedge\!\!\!\cdot\, \big[\text{s-event:}\big[\text{e=}u\text{:}T_1\big]\big]$

(19) says that an agent with lexical resource Lex(T, C) who judges a speech event, u, to be of type T is licensed to judge that there is a sign of type Lex(T, C) whose 's-event.e'-field contains u.

Strings of utterances of words can be classified as utterances of phrases. That is, speech events are hierarchically organized into types of speech events in the way that we discussed at the beginning of this chapter. Agents have resources which allow them to reclassify a string of signs of certain types ("the daughters") into a single sign of another type ("the mother"). So for example a string of type *Det⌢N* can lead us to the conclusion that we have observed a sign of type *NP* whose daughters are of the type *Det⌢N*. The resource that allows us to do this is a rule which we will model as the function in (20a) which we will represent as (20b).

(20) a. λu : *Det⌢N* .
NP $\wedge\!\!\!\cdot\, \big[\text{syn:}\big[\text{daughters=}u\text{:}\textit{Det⌢N}\big]\big]$

b. RuleDaughters(*NP*, *Det⌢N*)

'RuleDaughters' is to be the function in (21).

(21) λT_1 : *Type*
λT_2 : *Type* .
$\lambda u : T_1 \,.\, T_2 \wedge\!\!\!\cdot\, \big[\text{syn:}\big[\text{daughters=}u\text{:}T_1\big]\big]$

Thus 'RuleDaughters', if provided with a subtype of *Sign*$^+$ and a subtype of *Sign* as arguments, will return a function which maps a string of signs of the first type to the second type with the restriction that the daughters field is filled by the string of signs. 'RuleDaughters' is one of a number of sign type construction operations which we will introduce as universal resources which have the property of returning what we will call a sign combination function. The licencing conditions associated with sign combination functions are as characterized in (22).

(22) If $f : (T_1 \rightarrow Type)$ is a sign combination function available to agent A, then for any u, $u :_A T_1$ licenses $:_A f(u)$

This means, for example, that if you categorize a string of signs as being of type *Det⌢N* then you can conclude that there is a sign of type *NP* with the additional restriction that its daughters are u.

'RuleDaughters' takes care of the 'daughters'-field but it says nothing about the 's-event.e'-field, that is the phonological type associated with the new sign. This should be required to be the concatenation of all the 's-event.e'-fields in the daughters. If $u : T^+$ where T is a record type containing the path π, we will use concat$_i(u[i].\pi)$, the concatenation of all the values $u[i].\pi$ for each element in the string u in the order in which they occur in the string. (This notation is made precise in Appendix A.13.) We can now formulate the function ConcatPhon as in (23)

(23) λu:[s-event:[e:*Phon*]]$^+$.
[s-event : [e=concat$_i(u[i]$.s-event.e) : *Phon*]]

ConcatPhon will map any string of speech events to the type of a single speech event whose phonology (that is the value of 's-event.e') is the concatenation of the phonologies of the individual speech events in the string.

We want to combine the function (23) with a function like that in (20). We do this by merging the domain types of the two functions and also merging the types that they return. This is shown in (24a) which in deference to standard linguistic notation for phrase structure rules could be represented as (24b).[4]

(24) a. λu : *Det⌢N* $\wedge$ [s-event:[e:*Phon*]]$^+$.
NP $\wedge$ [syn:[daughters=u:*Det⌢N*]]
$\wedge$ [s-event:[e=concat$_i(u[i]$.s-event.e):*Phon*]]

b. *NP* $\longrightarrow$ *Det N*

[4]Note that '$\longrightarrow$' used in the phrase structure rule in (24b) is not the same arrow as '$\rightarrow$' which is used in our notation for function types. We trust that the different contexts in which they occur will help to distinguish them.

In general we say that if $C, C_1, \ldots, C_n$ are category sign types as in (13) then $C \longrightarrow C_1 \ldots C_n$ represents RuleDaughters(C, $C_1 ^\frown \ldots ^\frown C_n$) $\stackrel{\wedge}{:}$ ConcatPhon where for any type returning functions $\lambda r : T_1 \ . \ T_2(r)$ and $\lambda r : T_3 \ . \ T_4(r)$ $\lambda r : T_1 \ . \ T_2(r) \stackrel{\wedge}{:} \lambda r : T_3 \ . \ T_4(r)$ denotes the function $\lambda r : T_1 \dot{\wedge} T_3 \ . \ T_2(r) \dot{\wedge} T_4(r)$. Thus the function in (24) can be represented in a third way as in (25).

(25) RuleDaughters(*NP*, *Det*$^\frown$*N*) $\stackrel{\wedge}{:}$ ConcatPhon

The hope is that the ability to factorize rules into "bite-size" components will enable us to build a theory of resources that will allow us to study them in isolation and will also facilitate the development of theories of learning. It gives us a clue to how agents can build new rules by combining existing components in novel ways. It has implications for universality as well. For example, while the rule *NP* $\longrightarrow$ *Det N* is not universal (though it may be shared by a large number of languages), ConcatPhon is a universally available rule component, allbeit a trivial universal which says that you can have concatenations of speech events to make a larger speech event.

The rules associated with our small grammar are given by (26)

(26) *S* $\longrightarrow$ *NP VP*
NP $\longrightarrow$ *Det N*
VP $\longrightarrow$ *V NP*

It may seem that we have done an awful of work to arrive at simple phrase structure rules. Some readers might wonder why it is worth all this trouble to ground the rules in a theory of events and action when what we come up with in the end is something that can be expressed in a standard notation which is one of the first things that a student of syntax learns. One reason has to do with our desire to explore the relationship between the perception and processing of non-linguistics events and speech events as discussed at the beginning of this chapter. Another reason has to do with placing natural constraints on syntax. By grounding syntactic structure in types of events we provide a motivation for the kind of discussion in Cooper (1982). An abstract syntax which proposes constituent structure which does not correspond to speech events is not grounded in the same way and thus presents a different kind of theory.

3.2 Semantics

We have so far specified our sign types in terms of phonology and syntax. Now we need to specify the content in the 'cnt'-field. We shall start by accounting for the contents of the lexical items

specified in (18). We consider first the common nouns *composer* and *conductor*. For each of these we introduce a predicate of arity $\langle Ind \rangle$. (See Appendix A.3.2 for discussion of predicates and arity.) Our universal resources will include a function, 'SemCommonNoun' which will construct a common noun content from such a predicate, p. This is defined as in (27).

(27) SemCommonNoun(p) = $\lambda r{:}\left[\text{x:}Ind\right] . \left[\ \text{e} \ : \ p(r.\text{x})\ \right]$

The function in (27) is of type ($\left[\text{x:}Ind\right] \rightarrow RecType$). That is, it is a function which maps any record containing a field labelled 'x' with an individual as value to a record type. We will abbreviate this type as *Ppty* (for "property") and we will call functions of this type *properties*. In our compositional semantics, properties will play a similar role as functions from individuals to truth values ($\langle e, t \rangle$) in Montague semantics. In place of individuals, we use records with an 'x'-field containing an individual. The motivation for this will become apparent in Chapter 4 when we discuss the temperature puzzle. In place of Montague's truth-values (that is, objects of Montague's type t) we use record types. Record types play the role of "propositions" in our system. Types, thought of as types of situations, can be considered as truth-bearing objects. They are true just in case there is something of the type and false otherwise, that is, if there is nothing of the type. The fact that we use the "proposition-like" objects as the results that our properties return is an essential ingredient in our intensional treatment of properties. In this way it follows in the tradition of property theory (Chierchia and Turner, 1988; Fox and Lappin, 2005) and Thomason's intensional approach to propositional attitudes (Thomason, 1980).

We can now combine the 'Lex'-function which builds sign types excluding content information with our new way of constructing common noun content. We define a function $\text{Lex}_{\text{CommonNoun}}$ which takes a phonological type and a predicate and returns a sign type. This is defined in (28).

(28) $\text{Lex}_{\text{CommonNoun}}(T_{\text{phon}}, p) =$
$\text{Lex}(T_{\text{phon}}, N) \wedge\!\!\!\cdot\ \left[\text{cnt=SemCommonNoun}(p)\text{:}Ppty\right]$

Note that the type of the content required here is *Ppty*. In Chapter 2 we defined the content type *Cnt* to be identical with *RecType*. Now we have to revise the definition of *Cnt* to be ($RecType \vee Ppty$). We will add further disjuncts to allow for more possibilities as we progress.

In order to cover the two common nouns *conductor* and *composer* we can include the sign types in (29) among our resources.

(29) a. $\text{Lex}_{\text{CommonNoun}}$("composer", composer)

b. $\text{Lex}_{\text{CommonNoun}}$("conductor", conductor)

Following Montague's (1973) original strategy we shall treat the contents of noun-phrases such as *Dudamel* or *a conductor* as being functions from properties to truth-bearing elements, that is, in our terms, record types. That is, noun-phrase contents will be of type (*Ppty→RecType*) which we will abbreviate as *Quant* (for "quantifier"). This means that we should now redefine the type of contents, *Cnt*, as *RecType* $\vee$ *Ppty* $\vee$ *Quant*.[5]

Dudamel and *Beethoven* will receive proper name contents. The recipe for constructing a proper name content based on a particular individual a is given by SemPropName(a) as defined in (30).

(30) SemPropName(a) =
λP:*Ppty* . $P(\left[\text{x=}a\right])$

We define $\text{Lex}_{\text{PropName}}$ which takes a phonological type (a name) and an individual (the referent of the name) and returns a sign type as in (31).

(31) $\text{Lex}_{\text{PropName}}(T_{\text{Phon}}, a)$ =
$\text{Lex}(T_{\text{Phon}}, \textit{NP}) \wedge\!\!\!\cdot\ \left[\text{cnt=SemPropName}(a)\text{:}\textit{Quant}\right]$

Resources to cover the proper names in our grammar could be as in (32) where d, b : *Ind* (two individuals, Dudamel and Beethoven).

(32) a. $\text{Lex}_{\text{PropName}}$("Dudamel", d)

b. $\text{Lex}_{\text{PropName}}$("Beethoven", b)

Note that there is nothing to prevent us from constructing sign types with the same phonological type but different contents. Thus proper names are not required to be "logically proper" in the sense that there is one and only one individual which can be referred to by an utterance belonging to the phonological type. Names can be ambiguous. For example, there are many composers named Bach and Strauss. We have the means to construct sign types for all of them on an as needed basis.

Now that we have both properties and quantifiers let us check at this point that we are on the right track for combining them in something like the kind of way that we will need for compositional semantics. Suppose we want to combine a proper name content for *Dudamel* (33a) with the property of being a conductor (33b). The obvious way to do this is by applying the function in

[5]Omitting parentheses for clarity.

(33a) to the argument (33b) as represented in (33c). According to the definition of functional application in Appendix A.4, (33c) is identical to (33d) which in turn is identical to (33e). In turn the dot notation for record path values defined in Appendix A.11 shows (33e) to be identical to (33f).

(33) a. λP:*Ppty* . $P(\left[\text{x}=d\right])$

b. λr:$\left[\text{x}:\textit{Ind}\right]$. $\left[\ \text{e} \ : \ \text{conductor}(r.\text{x})\ \right]$

c. λP:*Ppty* . $P(\left[\text{x}=d\right])$
$(\lambda r$:$\left[\text{x}:\textit{Ind}\right]$. $\left[\ \text{e} \ : \ \text{conductor}(r.\text{x})\ \right])$

d. λr:$\left[\text{x}:\textit{Ind}\right]$. $\left[\ \text{e} \ : \ \text{conductor}(r.\text{x})\ \right](\left[\text{x}=d\right])$

e. $\left[\ \text{e} \ : \ \text{conductor}(\left[\text{x}=d\right].\text{x})\ \right]$

f. $\left[\ \text{e} \ : \ \text{conductor}(d)\ \right]$

This means that if we were dealing with a language like Russian where *Dudamel is a conductor* corresponds to a proper name followed by a common noun we would have a good way of combining the two contents by applying the content of the proper name to the content of the common noun.[6] However, things are not quite so straightforward in English. Here we use an indefinite article to form the noun phrase *a conductor*. We shall treat the content of indefinite articles as a function that maps properties to quantifiers involving the existential relation between properties. That is, it will be a function of type (*Ppty*→*Quant*), a type which should be added to our definition of *Cnt* which now becomes *RecType* ∨ *Ppty* ∨ *Quant* ∨ (*Ppty*→*Quant*). As part of our universal resources we introduce a function 'SemIndefArt' which is defined as the function in (34).

(34) λQ:*Ppty* .
λP:*Ppty* . $\left[\begin{array}{lcl} \text{restr}=Q & : & \textit{Ppty} \\ \text{scope}=P & : & \textit{Ppty} \\ \text{e} & : & \text{exist(restr, scope)} \end{array}\right]$

We can also define a universal resource, $\text{Lex}_{\text{IndefArt}}$, which associates a phonological type (corresponding to an indefinite article in the language) with this content, as defined in (35).

[6] An alternative would be to treat the content of a proper name as a record rather than a quantifier and apply the property to the record as in (33d). This would correspond to the treatment of proper names as individual denoting as discussed, for example, by Partee (1986).

(35) $\text{Lex}_{\text{IndefArt}}(T_{\text{Phon}})$ =
$\text{Lex}(T_{\text{Phon}}, \textit{Det}) \mathbin{\dot{\wedge}} \left[\text{cnt=SemIndefArt:}(\textit{Ppty}\rightarrow\textit{Quant})\right]$

The local resource for the English indefinite article would thus be (36).

(36) $\text{Lex}_{\text{IndefArt}}$("a")

The compositional semantics of a noun-phrase consisting of a determiner followed by a noun will be the content of the determiner applied to the content of the noun. This is a case of *content forward application*. We define a function 'CntForwardApp', which is part of the universal resources, as in (37).

(37) λT_1:*Type* λT_2:*Type* .
λu:$\left[\text{cnt:}(T_2 \rightarrow T_1)\right]^\frown\left[\text{cnt:}T_2\right]$.
$\left[\text{cnt=}u\text{[0].cnt(}u\text{[1].cnt):}T_1\right]$

The intuition behind this function is that if you observe a string of two utterances, the first of which has a content of type $(T_2 \rightarrow T_1)$ and the second of which has a content of type T_2 then you are licensed to conclude that there is an utterance whose content is the result of applying the content of the first element in the string to the content of the second element of the string. (For the notation $s[n]$ representing the nth element of a string s see Appendix A.13.) We can use 'CntForwardApp' to add constraints on content to a phrase structure rule as in the example in (38).

(38) *NP* $\longrightarrow$ *Det N* $\mathbin{\dot{\wedge}}$ CntForwardApp(*Ppty*,*Quant*)

Recall from (24a) that *NP* $\longrightarrow$ *Det N* is the function (39a). CntForwardApp(*Ppty*,*Quant*) is the function (39b). Merging these two functions yields (39c).

(39) a. λu : *Det*$^\frown$*N* $\wedge\!\!\!\cdot$ $\big[$s-event:$\big[$e:*Phon*$\big]\big]^+$.
NP $\wedge\!\!\!\cdot$ $\big[$syn:$\big[$daughters=u:*Det*$^\frown$*N*$\big]\big]$
$\wedge\!\!\!\cdot$ $\big[$s-event:$\big[$e=concat$_i$($u[i]$.s-event.e):*Phon*$\big]\big]$

b. λu:$\big[$cnt:$(Ppty \rightarrow Quant)\big]^\frown\big[$cnt:*Ppty*$\big]$.
$\big[$cnt=u[0].cnt(u[1].cnt):*Quant*$\big]$

c. λu : *Det*$^\frown$*N* $\wedge\!\!\!\cdot$ $\big[$s-event:$\big[$e:*Phon*$\big]\big]^+$
$\wedge\!\!\!\cdot$ $\big[$cnt:$(Ppty \rightarrow Quant)\big]^\frown\big[$cnt:*Ppty*$\big]$.
NP $\wedge\!\!\!\cdot$ $\big[$syn:$\big[$daughters=u:*Det*$^\frown$*N*$\big]\big]$
$\wedge\!\!\!\cdot$ $\big[$s-event:$\big[$e=concat$_i$($u[i]$.s-event.e):*Phon*$\big]\big]$
$\wedge\!\!\!\cdot$ $\big[$cnt=u[0].cnt(u[1].cnt):*Quant*$\big]$

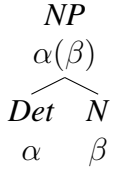
A convenient abbreviatory notation for this interpreted phrase structure rule is given in (40).

(40) $S \longrightarrow Det\ N \mid Det'(N')$

Here *Det*′ and *N*′ represent the contents of the determiner and noun.

We can represent the type (41a) using an informal diagrammatic tree notation which is common in linguistics as in (41b).[7]

(41) a. *NP* $\wedge\!\!\!\cdot$ $\begin{bmatrix} \text{s-event:}\big[\text{e=syn.daughters[0].s-event.e}^\frown\text{syn.daughters[0].s-event.e:}\textit{Phon}\big] \\ \text{syn:}\big[\text{daughters:}\textit{Det}^\frown\textit{N}\big] \\ \text{cnt=syn.daughters[0].cnt(syn.daughters[1].cnt):}\textit{Quant} \end{bmatrix}$

b.


Here what is written under the category type (e.g. α, β) represents the value in the 'cnt'-field.

The content of an utterance of *a conductor* will be (42a) applied to (42b), that is (42c).

[7] A similar use of tree notation, though relating to typed feature structures rather than types, is used in HPSG (see, for example, Ginzburg and Sag, 2000, Chapter 2).

(42) a. λQ:*Ppty* .

$$\lambda P\text{:}\textit{Ppty}\ .\ \left[\begin{array}{lcl} \text{restr}=Q & : & \textit{Ppty} \\ \text{scope}=P & : & \textit{Ppty} \\ \text{e} & : & \text{exist(restr, scope)} \end{array}\right]$$

b. $\lambda r\text{:}\left[\text{x:}\textit{Ind}\right]\ .\ \left[\ \text{e} \ : \ \text{conductor}(r.\text{x})\ \right]$

c. λP:*Ppty* .

$$\left[\begin{array}{lcl} \text{restr}=\lambda r\text{:}\left[\text{x:}\textit{Ind}\right]\ .\ \left[\ \text{e} \ : \ \text{conductor}(r.\text{x})\ \right] & : & \textit{Ppty} \\ \text{scope}=P & : & \textit{Ppty} \\ \text{e} & : & \text{exist(restr, scope)} \end{array}\right]$$

We will now look in more detail at the nature of the generalized quantifier in (42c). 'exist' is a predicate with arity $\langle$*Ppty*,*Ppty*$\rangle$, that is, it corresponds to a relation between two properties. The classical account of generalized quantifiers (Barwise and Cooper, 1981; Peters and Westerståhl, 2006, and much other literature) treats such quantifier relations as relations between sets. Here we will follow Cooper (2011, 2013) in relating our treatment directly to the classical relation between sets, although, as argued in Cooper (2012a) based on earlier work by Keenan and Stavi (1986), there are ultimately good reasons for exploiting the intensionality of properties. If P is a property the relevant set is the set of individuals which have the property, which we will represent as $[\downarrow P]$. This is defined as in (43) where we use the notation $\check{}[T]$ to represent $\{a \mid a : T\}$.

(43) $[\downarrow P] = \{a \mid \exists r[r : \left[\text{x:}\textit{Ind}\right] \wedge r.\text{x} = a \wedge \check{}[P(r)] \neq \emptyset]\}$

Following the terminology of Cooper (2011, 2013) we will call $[\downarrow P]$ the property extension, or *P-extension*, of property P. If P and Q are properties we want exist(P, Q) to be a type of situations which will be non-empty (that is, "true") just in case the P-extensions of P and Q have a non-empty overlap, that is there is some individual which has both property P and property Q. In symbols we can express this as (44).

(44) $\check{}[\text{exist}(P, Q)] \neq \emptyset$ iff $[\downarrow P] \cap [\downarrow Q] \neq \emptyset$

This places a requirement on objects which are assigned to the type 'exist(P, Q)' without actually tying down what kind of object they have to be. That is, it leaves it open as to which objects get assigned to the type, as long as they respect this requirement. It places a constraint on F in the models discussed on p. 7. We can, however, go a step further and make precise exactly which objects these should be. One intution is that a situation e should be of type exist(P, Q) just in case it is a witness (or "proof") of the fact that the "exist"-relation holds between P and Q (that is, that the P-extensions of P and Q have a non-empty overlap). Another intuition is that a witness for the type would be the pair $\langle P, Q\rangle$ just in case the "exist"-relation holds between P

and Q. This is reminiscent of the way in which Σ-types are treated in constructive type theory (as discussed on p. 8). There is a way of combining these two intuitions. We model situations as records and we can model a pair as a record with two fields containing the respective members of the pair. Thus we can think of the pair as a situation which has the two properties in appropriate roles. Our definition is given in (45).[8]

(45) e : exist(P, Q) iff $e : \begin{bmatrix} \text{arg}_1\text{=}P\text{:}\textit{Ppty} \\ \text{arg}_2\text{=}Q\text{:}\textit{Ppty} \end{bmatrix}$ and $[\downarrow P] \cap [\downarrow Q] \neq \emptyset$

Let us consider what we get when we apply the content we have for *a conductor*, (46a) (repeated from (42c)), to the property of composing, (46b). The result which would correspond to *a conductor composes* if we were to introduce *composes* as an intransitive verb in our resources, is given in (46c).

(46)

a. $\lambda P\text{:}\textit{Ppty}\,.\, \left[\begin{array}{lll} \text{restr=}\lambda r\text{:}\left[\text{x:}\textit{Ind}\right]\,.\,\left[\begin{array}{lll}\text{e} & : & \text{conductor}(r.\text{x})\end{array}\right] & : & \textit{Ppty} \\ \text{scope=}P & : & \textit{Ppty} \\ \text{e} & : & \text{exist(restr, scope)} \end{array}\right]$

b. $\lambda r\text{:}\left[\text{x:}\textit{Ind}\right]\,.\,\left[\begin{array}{lll}\text{e} & : & \text{compose}(r.\text{x})\end{array}\right]$

c. $\left[\begin{array}{lll} \text{restr=}\lambda r\text{:}\left[\text{x:}\textit{Ind}\right]\,.\,\left[\begin{array}{lll}\text{e} & : & \text{conductor}(r.\text{x})\end{array}\right] & : & \textit{Ppty} \\ \text{scope=}\lambda r\text{:}\left[\text{x:}\textit{Ind}\right]\,.\,\left[\begin{array}{lll}\text{e} & : & \text{compose}(r.\text{x})\end{array}\right] & : & \textit{Ppty} \\ \text{e} & : & \text{exist(restr, scope)} \end{array}\right]$

What would it mean for there to be something of type (46c)? In other words, what would be required to make the sentence *a conductor composes* true? There would have to be a record which contains the three fields in the record in (47) and which meets the condition indicated.

[8]Note that this runs dangerously close to Russell's paradox. In Appendix A.3.2 we say that ptypes are labelled sets. The ptype we represent as exist(P,Q) would be the labelled set $\{\langle \text{pred}, \text{exist}\rangle, \langle \text{arg}_1, P\rangle, \langle \text{arg}_2, Q\rangle\}$. Records are also labelled sets. The only thing that stops this labelled set from being a record and therefore according to (45) allowing for the possibility that exist(P,Q) : exist(P,Q) is that predicates are not technically assigned to a type according to our definitions. They are considered as type constructors but not themselves objects of a type. There are a number of other ways to avoid the problem if this way were to be deemed unacceptable for some reason. We just flag here, that however one constructs ptypes and the objects that are of ptypes, we should probably avoid allowing ptypes to be of themselves. Doing so, would not only run the risk of paradox, but would also not make much intuitive sense.

(47)
$$\left[\begin{array}{lcl} \text{restr} & = & \lambda r{:}\left[\text{x:}\mathit{Ind}\right] . \left[\begin{array}{lcl} \text{e} & : & \text{conductor}(r.\text{x}) \end{array}\right] \\ \text{scope} & = & \lambda r{:}\left[\text{x:}\mathit{Ind}\right] . \left[\begin{array}{lcl} \text{e} & : & \text{compose}(r.\text{x}) \end{array}\right] \\ \text{e} & = & \left[\begin{array}{lcl} \text{arg}_1 & = & \lambda r{:}\left[\text{x:}\mathit{Ind}\right] . \left[\begin{array}{lcl} \text{e} & : & \text{conductor}(r.\text{x}) \end{array}\right] \\ \text{arg}_2 & = & \lambda r{:}\left[\text{x:}\mathit{Ind}\right] . \left[\begin{array}{lcl} \text{e} & : & \text{compose}(r.\text{x}) \end{array}\right] \end{array}\right] \end{array}\right]$$
where the P-extensions of e.arg$_1$ and e.arg$_2$ have a non-empty overlap.

Given the availability of appropriate labels and predicates, the type theory will guarantee that a record of the form in (47) will exist, but it will not necessarily guarantee that the condition is met. This will depend on what is assigned to the basic types and ptype, that is, it will depend on the model.

This gives us a version of the classical treatment of indefinite articles as involving the existential quantifier, expressed in terms of a generalized quantifier which compares sets. There is, of course, a real and important question whether this is an appropriate content for the sentence *a conductor composes* which tends to get a generic reading something like "conductors, in general, compose". We will return to this issue in Chapter 4 where we will deal with the indefinite article in more detail. For now, we will ignore the sentence *a conductor composes* since we are not considering syntactic resources for it anyway.

We are concerned with finding a way to interpret the verb phrase *is a conductor*. Can we find a content for *is* which could be combined with the content for *a conductor* given in (42c) to produce an appropriate interpretation for the verb-phrase? Montague's (1973) strategy for assigning a content to *is* is reproduced in our terms in (48).

(48) $\lambda\mathcal{Q}$:*Quant* .
$\lambda r_1{:}\left[\text{x:}\mathit{Ind}\right]$.
$\mathcal{Q}(\lambda r_2{:}\left[\text{x:}\mathit{Ind}\right] . \left[\begin{array}{lcl} \text{e} & : & r_2.\text{x} = r_1.\text{x} \end{array}\right])$

Here we take '=' to be a predicate (used in infix notation, that is $a = b$ to mean $= (a, b)$) with arity $\langle$*Ind*,*Ind*$\rangle$.[9] The conditions for being of this type could be expressed as (49).

(49) $e : a = b$ iff $e : \begin{bmatrix} \text{arg}_1{=}a{:}\mathit{Ind} \\ \text{arg}_2{=}b{:}\mathit{Ind} \end{bmatrix}$ and a is identical with b

We will call (48) 'SemBe'. It will be included among the universal resources, together with the 'Lex$_{\text{be}}$' as defined in (50).

[9] Actually, we would want '=' to be polymorphic and assign it the set of arities $\{T \in \textbf{Type} \mid \langle T, T\rangle\}$

(50) If T_{Phon} is a phonological type, then $\mathrm{Lex}_{\mathrm{be}}(T_{\mathrm{Phon}})$ is $\mathrm{Lex}(T_{\mathrm{Phon}}, V) \wedge \left[\text{cnt=SemBe:}(\textit{Quant}\rightarrow\textit{Ppty})\right]$

Among the lexical resources for English we have $\mathrm{Lex}_{\mathrm{be}}$("is").

Now let us see what we get when we combine (48) with the content of *a conductor*. This involves applying (48), repeated as (51a), to (42c), repeated as (51b). The result of this application is (51c).

(51) a. $\lambda\mathcal{Q}$:*Quant* .
λr_1:$\left[\text{x:}\textit{Ind}\right]$.
$\mathcal{Q}(\lambda r_2$:$\left[\text{x:}\textit{Ind}\right]$. $\left[\ \text{e} \ : \ r_2.\text{x} = r_1.\text{x}\ \right])$

b. λP:*Ppty* . $\left[\begin{array}{lll} \text{restr=}\lambda r\text{:}\left[\text{x:}\textit{Ind}\right] . \left[\ \text{e} \ : \ \text{conductor}(r.\text{x})\ \right] & : & \textit{Ppty} \\ \text{scope=}P & : & \textit{Ppty} \\ \text{e} & : & \text{exist(restr, scope)} \end{array}\right]$

c. λr_1:$\left[\text{x:}\textit{Ind}\right]$.
$\left[\begin{array}{lll} \text{restr=}\lambda r\text{:}\left[\text{x:}\textit{Ind}\right] . \left[\ \text{e} \ : \ \text{conductor}(r.\text{x})\ \right] & : & \textit{Ppty} \\ \text{scope=}\lambda r_2\text{:}\left[\text{x:}\textit{Ind}\right] . \left[\ \text{e} \ : \ r_2.\text{x} = r_1.\text{x}\ \right] & : & \textit{Ppty} \\ \text{e} & : & \text{exist(restr, scope)} \end{array}\right]$

In order to obtain a content for *Dudamel is a conductor* we apply the content of *Dudamel*, (33a), repeated as (52a), to (51c), repeated as (52b), with result (52c).

(52) a. λP:*Ppty* . $P(\left[\text{x=}d\right])$

b. λr_1:$\left[\text{x:}\textit{Ind}\right]$.
$\left[\begin{array}{lll} \text{restr=}\lambda r\text{:}\left[\text{x:}\textit{Ind}\right] . \left[\ \text{e} \ : \ \text{conductor}(r.\text{x})\ \right] & : & \textit{Ppty} \\ \text{scope=}\lambda r_2\text{:}\left[\text{x:}\textit{Ind}\right] . \left[\ \text{e} \ : \ r_2.\text{x} = r_1.\text{x}\ \right] & : & \textit{Ppty} \\ \text{e} & : & \text{exist(restr, scope)} \end{array}\right]$

c. $\left[\begin{array}{lll} \text{restr=}\lambda r\text{:}\left[\text{x:}\textit{Ind}\right] . \left[\ \text{e} \ : \ \text{conductor}(r.\text{x})\ \right] & : & \textit{Ppty} \\ \text{scope=}\lambda r_2\text{:}\left[\text{x:}\textit{Ind}\right] . \left[\ \text{e} \ : \ r_2.\text{x} = d\ \right] & : & \textit{Ppty} \\ \text{e} & : & \text{exist(restr, scope)} \end{array}\right]$

The type (52c) is distinct from the type (33f), repeated as (53), which we obtained by applying the content of *Dudamel* directly to the content of *conductor*.

(53) $\left[\ \text{e} \ : \ \text{conductor}(d)\ \right]$

There is, however, an equivalence that holds between (52c) and (53). The equivalence is not that they share the same set of witnesses. We can characterize the set of witnesses of (52c) and (54a) and the witnesses of (53) as (54b).

(54)

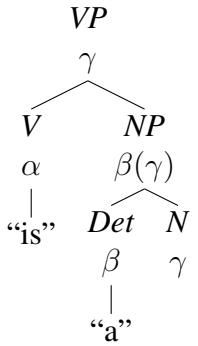
a. $\{ \left[\begin{array}{lcl} \text{restr} & = & P \\ \text{scope} & = & Q \\ \text{e} & = & \left[\begin{array}{lcl} \text{arg}_1 & = & P \\ \text{arg}_2 & = & Q \end{array}\right] \end{array}\right] \mid$
$P = \lambda r{:}\left[\text{x:}\textit{Ind}\right] . \left[\begin{array}{lcl} \text{e} & : & \text{conductor}(r.\text{x}) \end{array}\right] \wedge$
$Q = \lambda r{:}\left[\text{x:}\textit{Ind}\right] . \left[\begin{array}{lcl} \text{e} & : & r.\text{x} = d \end{array}\right] \wedge$
$[\downarrow P] \cap [\downarrow Q] \neq \emptyset\}$

b. $\{\left[\begin{array}{lcl} \text{e} & = & s \end{array}\right] \mid s : \text{conductor}(d)\}$

The sets in (54) do not have any members in common. The equivalence is a weaker "truth-conditional" equivalence. (52c) has a witness ("is true") if and only if (53) has a witness. This is because the P-extensions of the property of being a conductor and the property of being identical with Dudamel can have a non-empty overlap if and only if Dudamel is a conductor. We might try to characterize the difference between the property associated with *conductor* and the property associated with *is a conductor* as "the property of being an x such that conductor(x)" and "the property of being an x such that there is a y such that conductor(y) and $y = x$". The two are truth-conditionally equivalent and for this reason in Montague's system they turn out to be the same property. For us, since we are taking a more intensional approach than Montague, they are distinct properties but they are nevertheless truth-conditionally equivalent.

Since we have two distinct properties, the question is raised whether the property that is associated with the verb-phrase should be the same as the property associated with the common noun or whether it should be the property proposed here involving existential quantification. One way to do this is to create a type corresponding to the tree in (55).

(55)



This is not compositional in the standard sense because the content of the verb phrase is not defined as some operation applied to the contents of the verb and the noun phrase, but rather it makes the content of the verb phrase be the content of the noun. Furthermore, it requires the verb and determiner utterances be of the specific types "is" and "a" respectively. This gives (55) the flavour of representing a construction type as discussed in a variety of approaches to Construction Grammar, see, for example, Boas and Sag (2012). We can allow the type corresponding to (55) by introducing the update function (56).

$$\text{(56)}\quad \lambda u{:}V \mathbin{\dot\wedge} \left[\text{s-event:}\left[\text{e:“is”}\right]\right] \frown NP \mathbin{\dot\wedge} \left[\text{syn:}\left[\begin{array}{r}\text{daughters:}Det \mathbin{\dot\wedge} \left[\text{s-event:}\left[\text{e:“a”}\right]\right] \\ \frown\ N \mathbin{\dot\wedge} \left[\text{cnt:}Ppty\right]\end{array}\right]\right]. \\ VP \mathbin{\dot\wedge} \left[\text{cnt=}u\text{[2].syn.daughters[2].cnt:}Ppty\right]$$

We can call this function CnstrIsA ("is-a construction") and merge it with $VP \longrightarrow V\ NP$. Thus one of the resources available for English is (57).

$$\text{(57)}\quad VP \longrightarrow V\ NP \mathbin{\dot\wedge} \text{CnstrIsA}$$

This suggests that a phrase structure and construction based approach can be combined within a single framework. Since we are working with a toy fragment where the only verb is *is* and the only determiner is *a*, we can make do with (57) as the only resource for assigning content to verb-phrases. In a more general grammar we would, of course, require in addition a rule that applies the content of the verb to the content of the object noun-phrase as in (58).

$$\text{(58)}\quad VP \longrightarrow V\ NP \mathbin{\dot\wedge} \text{CntForwardApp}(Quant, Ppty)$$

Allowing both resources (57) and (58) simultaneously raises the issue of what the relationship should be between them. Should the more specific rule (57) take precedence and guarantee that the only content associated with the verb phrase *is a conductor* is the property which is the content of *conductor*? Or should the verb phrase be ambiguous between this interpretation and the property obtained by applying the content of *is* to the content of *a conductor*?

A more pressing issue, perhaps, is what to do about the sentence in (59).

(59) #A conductor is Dudamel

We have used the marking '#' in (59) to indicate that an utterance of this sentence would under most, if not all, circumstances be considered to be odd, though it is difficult to rule it out as

ungrammatical, particularly if we are to use something corresponding to context-free phrase structure rules as we are. The oddness of (59) may have something to do with the tendency to interpret noun phrases with indefinite articles in subject position as generic as in (60).

(60) A conductor is a high-ranking individual in the musical hierarchy

(59) can be improved without becoming generic. Examples are given in (61).

(61) a. A conductor to reckon with is Dudamel

b. A conductor to consider is Dudamel

c. A conductor who impresses me as a leader in his generation is Dudamel

d. A conductor I would like to see more often in Gothenburg is Dudamel

This raises a lot of issues which we do not currently have tools to deal with. There is, however, something we can say, if we choose to allow the is-a construction interpretation of *is a conductor*. The content of the sentence *Dudamel is a conductor* on the construction analysis becomes (53), repeated as (62a), rather than (52c), repeated as (62b).

(62) a. $\left[\ \text{e} \ : \ \text{conductor}(d)\ \right]$

b. $\left[\begin{array}{lll} \text{restr}=\lambda r{:}\left[\text{x}{:}\textit{Ind}\right] . \left[\ \text{e} \ : \ \text{conductor}(r.\text{x})\ \right] & : & \textit{Ppty} \\ \text{scope}=\lambda r_2{:}\left[\text{x}{:}\textit{Ind}\right] . \left[\ \text{e} \ : \ r_2.\text{x} = d\ \right] & : & \textit{Ppty} \\ \text{e} & : & \text{exist(restr, scope)} \end{array}\right]$

If we include the resource (58) then the content of *is Dudamel* is (63a) applied to (63b), that is, (63c).

(63) a. $\lambda\mathcal{Q}{:}\textit{Quant} .$
$\lambda r_1{:}\left[\text{x}{:}\textit{Ind}\right] .$
$\mathcal{Q}(\lambda r_2{:}\left[\text{x}{:}\textit{Ind}\right] . \left[\ \text{e} \ : \ r_2.\text{x} = r_1.\text{x}\ \right])$

b. $\lambda P{:}\textit{Ppty} . P(\left[\text{x}{=}d\right])$

c. $\lambda r_1{:}\left[\text{x}{:}\textit{Ind}\right] . \left[\ \text{e} \ : \ d = r_1.\text{x}\ \right])$

The content of *A conductor is Dudamel* is (64a) applied to (64b) (identical with (63c)), which is (64c).

(64)

a. λP:*Ppty* . $\left[\begin{array}{lll} \text{restr}=\lambda r{:}\left[\text{x}{:}\mathit{Ind}\right] . \left[\ \text{e} : \text{conductor}(r.\text{x})\ \right] & : & \mathit{Ppty} \\ \text{scope}=P & : & \mathit{Ppty} \\ \text{e} & : & \text{exist(restr, scope)} \end{array}\right]$

b. $\lambda r_1{:}\left[\text{x}{:}\mathit{Ind}\right] . \left[\ \text{e} : d = r_1.\text{x}\ \right])$

c. $\left[\begin{array}{lll} \text{restr}=\lambda r{:}\left[\text{x}{:}\mathit{Ind}\right] . \left[\ \text{e} : \text{conductor}(r.\text{x})\ \right] & : & \mathit{Ppty} \\ \text{scope}=\lambda r_1{:}\left[\text{x}{:}\mathit{Ind}\right] . \left[\ \text{e} : d = r_1.\text{x}\ \right]) & : & \mathit{Ppty} \\ \text{e} & : & \text{exist(restr, scope)} \end{array}\right]$

(64c) is almost exactly the same type as (62b). The difference between them is that d is the first argument to the equality predicate in (64c) whereas in (62b) it is the second argument. Thus while an analysis that only the content of *is* that is based on Montague's original interpretation does predict different contents for *Dudamel is a conductor* and *a conductor is Dudamel*, the difference between the types hardly seems enough to explain the difference in reaction we have to the two sentences. Given the construction analysis for *Dudamel is a conductor* we get a markedly different type (62a) which does not involve existential quantification (even though it is truth conditionally equivalent to both the types with existential quantification). The only way that (62a) can be expressed according to the resources that we have developed in this chapter is by the sentence *Dudamel is a conductor*, using the non-compositional construction 'CnstrIsA'. Thus if (62a) is the target content and we do not wish to express a content involving existential quantification, *a conductor is Dudamel* is not an option.

We thus have the beginnings of an explanation of the difference in acceptability between the two sentences. It is not the whole story since we have not explained why the quantificational readings appear odd in these cases. Note that the distinction we are making between a non-quantified reading and a reading involving an existential quantification is not available on Montague's 1973 original approach since the fact that the two contents are truth-conditionally equivalent means for Montague that they are identical. The same holds for the kind of analysis discussed in Partee (1986) where even though the content may not be built up using existential quantification the final result is still the same content that would be expressed by using existential quantification because of the truth-conditional equivalence. One might try to introduce the distinction we are making by relating utterances to an expression in an artificial logical language in addition to the content. This would correspond to the notion of logical form as discussed for example by Heim and Kratzer (1998) and much current work in linguistic semantics. The idea might be that there are two distinct logical forms such as (65) which correspond to identical contents in Montague's terms.

(65) a. conductor(dudamel)

b. $\exists x$ [conductor(x) $\wedge$ x=dudamel]

Here the challenge would be to give an explanatory account of why one expression in an artificial language (65a) should be preferred over another (65b) when they both express the same content. An alternative is to follow Lewis (1972) (further developed by Cresswell, 1985). The idea here is that we keep a record not only of the final content but the way in which that content is constructed – that is we keep a record of the content of each of the syntactic constituents of the English sentence and the way these contents are combined. This idea, which goes back to the notion of intensional isomorphism introduced by Carnap (1956), provides enough structure to make the distinction required here. However, there are other problems with the proposal which we will take up in Chapter 4 when we discuss intensionality.

Since acknowledgements like *aha* and *ok* do not have a specified content (*cf.* the function 'sign$_{uc}$' we used for these words in Chapter 2), we do not need a function that specifies their content but can make do with the function 'Lex' which associates them with a sign type in which the content is unspecified.

3.3 Building a chart type

In Chapter 2 we made the simplifying assumption that a chart was a sign. Now we have a grammar we need to complicate this picture. We will present here a version of chart parsing as it is used in computational linguistics. For a recent textbook introduction to chart parsing see Jurafsky and Martin (2009), Chap. 13. The idea of a chart is that it should store all the hypotheses that we make during the processing of an utterance and allow us to compute new hypotheses to be added to the chart on the basis of what is already present in the chart. We will say that a chart is a record and we will use our resources to compute a chart type on the basis of utterance events. We will first go through an example of the incremental construction of a chart type for an agent processing an utterance of the sentence *Dudamel is a conductor*. Then we will consider what kind of update functions are needed in order to achieve this. We will, as usual, make the simplifying assumption that what we have at bottom is a string of word utterances as we are not dealing with the details of phonology. Thus we are giving a simplified view of incremental processing at the word level.

Suppose that we have so far heard an utterance of the word *Dudamel*. At this point we will say that the type of the chart is (66).

(66) $\begin{bmatrix} e_1 & : & \text{“Dudamel”} \\ e & : & [e_1\text{:start}(e_1)] \frown [e_1\text{:end}(e_1)] \end{bmatrix}$

The main event of the chart type (represented by the e-field) breaks the phonological event of type “Dudamel” down into a string of two events, the start and the end of the “Dudamel”-event.[10] Thus (66) records that we have observed an event of the phonological type “Dudamel” and an event consisting of the start of that event followed by the end of that event. Given that we have the resource the resource $\text{Lex}_{\text{PropName}}$(“Dudamel”, d) available (see Appendix B), we can update (66) to (67).

(67) $\begin{bmatrix} e_1 & : & \text{“Dudamel”} \\ e_2 & : & \text{Lex}_{\text{PropName}}(\text{“Dudamel”}, d) \wedge\!\!\!\cdot\ [\text{s-event:}[e{=}e_1\text{:}\textit{Phon}]\ ] \\ e & : & \begin{bmatrix} e_1\text{:start}(e_1) \\ e_2\text{:start}(e_2) \end{bmatrix} \frown \begin{bmatrix} e_1\text{:end}(e_1) \\ e_2\text{:end}(e_2) \end{bmatrix} \end{bmatrix}$

That is, we add the information to the chart that there is an event (labelled ‘e_2’) of the type which is the sign type corresponding to “Dudamel” and that the event which is the speech event referred to in that sign type is the utterance event, labelled by ‘e_1’. Furthermore the duration of the event labelled ‘e_2’ is the same as that labelled ‘e_1’. One could discuss where there are two events which are contemporaneous or whether there is a single utterance event which is of both types. The fact that we have presented two fields labelled ‘e_1’ and ‘e_2’ does not of itself prevent the two fields containing the same event. However, the fact that we have analyzed the sign as containing the speech event as a part (corresponding to the basic intuition that signs are pairings of utterances and contents) decides the issue for us. A sign is a record (a labelled set) which models a situation and we are not allowing sets to be members of themselves. Thus records cannot be a part of themselves.[11]

The type $\text{Lex}_{\text{PropName}}$(“Dudamel”, d) is a subtype of *NP*. Thus the event labelled ‘e_2’ could be the first item in a string that would be appropriate for the function which we have abbreviated as (68a) (see Appendix B) which has the type (68b).

(68) a. $S \longrightarrow NP\ VP \mid NP'(VP')$

b. $(NP^\frown VP \rightarrow \textit{Type})$

Thus in a way that is similar to the prediction by the dog in Chapter 1 that it should run after the stick which is help up and the kind of event that this will contribute to is a game of fetch so on

[10] These starting and ending events correspond to what are standardly called *vertices* in the chart parsing literature.

[11] ‘e_1’ and ‘e_2’ correspond to what are known as *passive edges* in the chart parsing literature. They represent information about potential constituents that have been found.

observing a noun-phrase event we can predict that it might be followed by a verb phrase event thus creating a sentence event. We will add a hypothesis event to our chart which takes place at the end of the noun-phrase event as in (69).[12]

(69)
$$\left[\begin{array}{lcl} e_1 & : & \text{“Dudamel”} \\ e_2 & : & \text{Lex}_{\text{PropName}}(\text{“Dudamel”}, d) \wedge\!\!\!\cdot\ \left[\text{s-event:}\left[\text{e=}\Uparrow^2\text{e}_1\text{:}\textit{Phon}\right]\ \right] \\ e_3 & : & \left[\begin{array}{l}\text{rule=}\textit{S} \longrightarrow \textit{NP VP} \mid \textit{NP}'(\textit{VP}')\text{:}(\textit{NP}^\frown\textit{VP} \rightarrow \textit{Type}) \\ \text{fnd=}\Uparrow\text{e}_2\text{:}\textit{Sign} \\ \text{req=}\textit{VP}\text{:}\textit{Type} \\ \text{e:required(req,rule)}\end{array}\right] \\ e & : & \left[\begin{array}{l}\text{e}_1\text{:start(e}_1\text{)} \\ \text{e}_2\text{:start(e}_2\text{)}\end{array}\right] \frown \left[\begin{array}{l}\text{e}_1\text{:end(e}_1\text{)} \\ \text{e}_2\text{:end(e}_2\text{)} \\ \text{e}_3\text{:start(e}_3\text{)}^\frown\text{end(e}_3\text{)}\end{array}\right] \end{array}\right]$$

In the e_3-field the ‘rule’-field is for a syntactic rule, that is, a function from a string of signs of a given type to a type. The ‘fnd’-field is for a sign or string of signs so far found which match an initial segment of a string of the type required by the rule. The ‘req’-field is the type of the remaining string required to satisfy the rule as expressed in the ‘e’-field. This hypothesis event both starts and ends at the end of the event of the noun-phrase event e_2.[13]

We can now progress to the next word in the input string as shown in (70).

(70)
$$\left[\begin{array}{lcl} e_1 & : & \text{“Dudamel”} \\ e_2 & : & \text{Lex}_{\text{PropName}}(\text{“Dudamel”}, d) \wedge\!\!\!\cdot\ \left[\text{s-event:}\left[\text{e=}\Uparrow^2\text{e}_1\text{:}\textit{Phon}\right]\ \right] \\ e_3 & : & \left[\begin{array}{l}\text{rule=}\textit{S} \longrightarrow \textit{NP VP} \mid \textit{NP}'(\textit{VP}')\text{:}(\textit{NP}^\frown\textit{VP} \rightarrow \textit{Type}) \\ \text{fnd=}\Uparrow\text{e}_2\text{:}\textit{Sign} \\ \text{req=}\textit{VP}\text{:}\textit{Type} \\ \text{e:required(req,rule)}\end{array}\right] \\ e_4 & : & \text{“is”} \\ e & : & \left[\begin{array}{l}\text{e}_1\text{:start(e}_1\text{)} \\ \text{e}_2\text{:start(e}_2\text{)}\end{array}\right] \frown \left[\begin{array}{l}\text{e}_1\text{:end(e}_1\text{)} \\ \text{e}_2\text{:end(e}_2\text{)} \\ \text{e}_3\text{:start(e}_3\text{)}^\frown\text{end(e}_3\text{)} \\ \text{e}_4\text{:start(e}_4\text{)}\end{array}\right] \frown \left[\text{e}_4\text{:end(e}_4\text{)}\right] \end{array}\right]$$

Note that the start of the “is”-event is aligned with the end of “Dudamel”-event. This allows for

[12] In terms of the traditional chart parsing terminology this corresponds to an *active edge* involving a *dotted rule*. The fact that the addition of this type to the chart type is triggered by finding something of an appropriate type to be the leftmost element in a string the would be an appropriate argument to the rule corresponds to what is called a *left-corner* parsing strategy.

[13] With respect to the word string event labelled by ‘e’, it is a *punctual* event.

the fact that there is no break between the words and that the exact pronunciation of the final /l/ in “Dudamel” is influenced by the pronuniciation of the initial /i/ in “is” through coarticulation.[14]

We can now go through similar procedures as we did for *Dudamel* adding both a lexical event based on our lexical resources and a hypothesis event based on the only rule for strings beginning with a V that we have in our resources. The result of these two steps is given in (71).

(71)

$$
\left[\begin{array}{lll}
\text{e}_1 & : & \text{“Dudamel”}\\
\text{e}_2 & : & \text{Lex}_{\text{PropName}}(\text{“Dudamel”}, d) \underset{\cdot}{\wedge} \left[\text{s-event:}\left[\text{e=}\Uparrow^2\text{e}_1{:}\textit{Phon}\right]\right]\\
\text{e}_3 & : & \left[\begin{array}{l}\text{rule=}S \longrightarrow NP\ VP \mid NP'(VP'){:}(NP^\frown VP \rightarrow \textit{Type})\\ \text{fnd=}\Uparrow \text{e}_2{:}\textit{Sign}\\ \text{req=}VP{:}\textit{Type}\\ \text{e:required(req,rule)}\end{array}\right]\\
\text{e}_4 & : & \text{“is”}\\
\text{e}_5 & : & \text{Lex}_{\text{be}}(\text{“is”})\underset{\cdot}{\wedge}\left[\text{s-event:}\left[\text{e=}\Uparrow^2 \text{e}_4{:}\textit{Phon}\right]\right]\\
\text{e}_6 & : & \left[\begin{array}{l} \text{rule=}VP \longrightarrow [_V\ \text{“is”}]\ [_{NP}\ [_{Det}\ \text{“a”}]\ N] \mid N'{:}\\ \qquad (V\underset{\cdot}{\wedge}\left[\text{s-event:}\left[\text{e:“is”}\right]\right]^\frown\\ \qquad NP\underset{\cdot}{\wedge}\left[\text{syn:}\left[\begin{array}{l}\text{daughters:}Det\underset{\cdot}{\wedge}\left[\text{s-event:}\left[\text{e:“a”}\right]\right]\\ \qquad {}^\frown N\underset{\cdot}{\wedge}\left[\text{cnt:}\textit{Ppty}\right]\end{array}\right]\right]\\ \qquad \rightarrow \textit{Type})\\ \text{fnd=}\Uparrow\text{e}_5{:}\textit{Sign}\\ \text{req=}NP{:}\textit{Type}\\ \text{e:required(req,rule)}\end{array}\right]\\
\text{e} & : & \left[\begin{array}{l}\text{e}_1{:}\text{start}(\text{e}_1)\\ \text{e}_2{:}\text{start}(\text{e}_2)\end{array}\right]^\frown \left[\begin{array}{l}\text{e}_1{:}\text{end}(\text{e}_1)\\ \text{e}_2{:}\text{end}(\text{e}_2)\\ \text{e}_3{:}\text{start}(\text{e}_3)^\frown\text{end}(\text{e}_3)\\ \text{e}_4{:}\text{start}(\text{e}_4)\\ \text{e}_5{:}\text{start}(\text{e}_5)\end{array}\right]^\frown \left[\begin{array}{l}\text{e}_4{:}\text{end}(\text{e}_4)\\ \text{e}_5{:}\text{end}(\text{e}_5)\\ \text{e}_6{:}\text{start}(\text{e}_6)^\frown\text{end}(\text{e}_6)\end{array}\right]
\end{array}\right]
$$

Now we can add *a* and *conductor* in a similar way with the result shown in (72).

[14]It also means that the number of elements in the string labelled ‘e’ is the same as the number of vertices in a standard chart.

(72)
$$\left[\begin{array}{lcl}
e_1 & : & \text{“Dudamel”} \\
e_2 & : & \text{Lex}_{\text{PropName}}(\text{“Dudamel”}, d) \mathbin{\dot\wedge} \left[\text{s-event:}\left[\text{e=}\Uparrow^2 e_1\text{:}\textit{Phon}\right]\right] \\
e_3 & : & \left[\begin{array}{l}\text{rule=}\textit{S} \longrightarrow \textit{NP VP} \mid \textit{NP}'(\textit{VP}')\text{:}(\textit{NP} \frown \textit{VP} \rightarrow \textit{Type}) \\ \text{fnd=}\Uparrow e_2\text{:}\textit{Sign} \\ \text{req=}\textit{VP}\text{:}\textit{Type} \\ \text{e:required(req,rule)}\end{array}\right] \\
e_4 & : & \text{“is”} \\
e_5 & : & \text{Lex}_{\text{be}}(\text{“is”}) \mathbin{\dot\wedge} \left[\text{s-event:}\left[\text{e=}\Uparrow^2 e_4\text{:}\textit{Phon}\right]\right] \\
e_6 & : & \left[\begin{array}{l}\text{rule=}\textit{VP} \longrightarrow [_{\textit{V}}\ \text{“is”}]\ [_{\textit{NP}}\ [_{\textit{Det}}\ \text{“a”}]\ \textit{N}] \mid \textit{N}\text{':} \\ \qquad (\textit{V} \mathbin{\dot\wedge} \left[\text{s-event:}\left[\text{e:“is”}\right]\right] \frown \\ \qquad\quad \textit{NP} \mathbin{\dot\wedge} \left[\text{syn:}\left[\begin{array}{l}\text{daughters:}\textit{Det} \mathbin{\dot\wedge} \left[\text{s-event:}\left[\text{e:“a”}\right]\right] \\ \qquad\qquad \frown\ \textit{N} \mathbin{\dot\wedge} \left[\text{cnt:}\textit{Ppty}\right]\end{array}\right]\right] \\ \qquad\quad \rightarrow \textit{Type}) \\ \text{fnd=}\Uparrow e_5\text{:}\textit{Sign} \\ \text{req=}\textit{NP}\text{:}\textit{Type} \\ \text{e:required(req,rule)}\end{array}\right] \\
e_7 & : & \text{“a”} \\
e_8 & : & \text{Lex}_{\text{IndefArt}}(\text{“a”}) \mathbin{\dot\wedge} \left[\text{s-event:}\left[\text{e=}\Uparrow^2 e_7\text{:}\textit{Phon}\right]\right] \\
e_9 & : & \left[\begin{array}{l}\text{rule=}\textit{NP} \longrightarrow \textit{Det N} \mid \textit{Det}'(\textit{N}')\text{:}(\textit{Det} \frown \textit{N} \rightarrow \textit{Type}) \\ \text{fnd=}\Uparrow e_8\text{:}\textit{Sign} \\ \text{req=}\textit{N}\text{:}\textit{Type} \\ \text{e:required(req,rule)}\end{array}\right] \\
e_{10} & : & \text{“conductor”} \\
e_{11} & : & \text{Lex}_{\text{CommonNoun}}(\text{“conductor”, conductor}) \mathbin{\dot\wedge} \left[\text{s-event:}\left[\text{e=}\Uparrow^2 e_{10}\text{:}\textit{Phon}\right]\right] \\
e & : & \left[\begin{array}{l}e_1\text{:start}(e_1) \\ e_2\text{:start}(e_2)\end{array}\right] \frown \left[\begin{array}{l}e_1\text{:end}(e_1) \\ e_2\text{:end}(e_2) \\ e_3\text{:start}(e_3) \frown \text{end}(e_3) \\ e_4\text{:start}(e_4) \\ e_5\text{:start}(e_5)\end{array}\right] \frown \left[\begin{array}{l}e_4\text{:end}(e_4) \\ e_5\text{:end}(e_5) \\ e_6\text{:start}(e_6) \frown \text{end}(e_6) \\ e_7\text{:start}(e_7) \\ e_8\text{:start}(e_8)\end{array}\right] \frown \\
 & & \qquad \left[\begin{array}{l}e_7\text{:end}(e_7) \\ e_8\text{:end}(e_8) \\ e_9\text{:start}(e_9) \frown \text{end}(e_9) \\ e_{10}\text{:start}(e_{10}) \\ e_{11}\text{:start}(e_{11})\end{array}\right] \frown \left[\begin{array}{l}e_{10}\text{:end}(e_{10}) \\ e_{11}\text{:end}(e_{11})\end{array}\right]
\end{array}\right]$$

Note that there is no possibility of adding a hypothesis event based on the utterance of *conductor* given the resources we have since our small grammar does not include a phrase structure rule for strings whose first element is of type *N*. However, now for the first time we have found something which fulfills one of our hypotheses. The hypothesis event labelled ‘e_9’ has the type *N* in its ‘req’-field. The event labelled ‘e_{11}’ is required to be of a subtype of *N* and thus fulfils the requirement of ‘e_9’. Furthermore, the start of e_{11} is aligned with the end (and also the start) of ‘e_9’. This means that we can update the chart-type by adding a new field for an event of the

type returned by applying 'e$_9$.rule' (a function) to the string e$_9$.fnd$^\frown$e$_{11}$. The start of this new *NP*-event will be aligned with the start of e$_9$.fnd (that is, e$_8$). The end of the new event is aligned with the end of e$_{11}$. The resulting chart-type is given in (73).

(73)
$$\left[\begin{array}{lcl}
e_1 & : & \text{“Dudamel”} \\
e_2 & : & \text{Lex}_{\text{PropName}}(\text{“Dudamel”}, d) \mathbin{\dot\wedge} \left[\text{s-event:}\left[\text{e=}\Uparrow^2 e_1\text{:}\textit{Phon}\right]\right] \\
e_3 & : & \left[\begin{array}{l}\text{rule=}\textit{S} \longrightarrow \textit{NP VP} \mid \textit{NP}'(\textit{VP}')\text{:}(\textit{NP}^\frown\textit{VP} \rightarrow \textit{Type}) \\ \text{fnd=}\Uparrow e_2\text{:}\textit{Sign} \\ \text{req=}\textit{VP}\text{:}\textit{Type} \\ \text{e:required(req,rule)}\end{array}\right] \\
e_4 & : & \text{“is”} \\
e_5 & : & \text{Lex}_{\text{be}}(\text{“is”}) \mathbin{\dot\wedge} \left[\text{s-event:}\left[\text{e=}\Uparrow^2 e_4\text{:}\textit{Phon}\right]\right] \\
e_6 & : & \left[\begin{array}{l}\text{rule=}\textit{VP} \longrightarrow [_{\textit{V}}\ \text{“is”}]\ [_{\textit{NP}}\ [_{\textit{Det}}\ \text{“a”}]\ \textit{N}] \mid \textit{N}'\text{:} \\ \qquad (\textit{V} \mathbin{\dot\wedge} \left[\text{s-event:}\left[\text{e:“is”}\right]\right]^\frown \\ \qquad\quad \textit{NP} \mathbin{\dot\wedge} \left[\text{syn:}\left[\begin{array}{l}\text{daughters:}\textit{Det} \mathbin{\dot\wedge} \left[\text{s-event:}\left[\text{e:“a”}\right]\right] \\ \qquad\qquad ^\frown \textit{N} \mathbin{\dot\wedge} \left[\text{cnt:}\textit{Ppty}\right]\end{array}\right]\right] \\ \qquad\quad \rightarrow \textit{Type}) \\ \text{fnd=}\Uparrow e_5\text{:}\textit{Sign} \\ \text{req=}\textit{NP}\text{:}\textit{Type} \\ \text{e:required(req,rule)}\end{array}\right] \\
e_7 & : & \text{“a”} \\
e_8 & : & \text{Lex}_{\text{IndefArt}}(\text{“a”}) \mathbin{\dot\wedge} \left[\text{s-event:}\left[\text{e=}\Uparrow^2 e_7\text{:}\textit{Phon}\right]\right] \\
e_9 & : & \left[\begin{array}{l}\text{rule=}\textit{NP} \longrightarrow \textit{Det N} \mid \textit{Det}'(\textit{N}')\text{:}(\textit{Det}^\frown\textit{N} \rightarrow \textit{Type}) \\ \text{fnd=}\Uparrow e_8\text{:}\textit{Sign} \\ \text{req=}\textit{N}\text{:}\textit{Type} \\ \text{e:required(req,rule)}\end{array}\right] \\
e_{10} & : & \text{“conductor”} \\
e_{11} & : & \text{Lex}_{\text{CommonNoun}}(\text{“conductor”, conductor}) \mathbin{\dot\wedge} \left[\text{s-event:}\left[\text{e=}\Uparrow^2 e_{10}\text{:}\textit{Phon}\right]\right] \\
e_{12} & : & e_9\text{.rule}(e_9\text{.fnd}^\frown e_{11}) \\
e & : & \left[\begin{array}{l}e_1\text{:start}(e_1) \\ e_2\text{:start}(e_2)\end{array}\right]^\frown \left[\begin{array}{l}e_1\text{:end}(e_1) \\ e_2\text{:end}(e_2) \\ e_3\text{:start}(e_3)^\frown\text{end}(e_3) \\ e_4\text{:start}(e_4) \\ e_5\text{:start}(e_5)\end{array}\right]^\frown \left[\begin{array}{l}e_4\text{:end}(e_4) \\ e_5\text{:end}(e_5) \\ e_6\text{:start}(e_6)^\frown\text{end}(e_6) \\ e_7\text{:start}(e_7) \\ e_8\text{:start}(e_8) \\ e_{12}\text{:start}(e_{12})\end{array}\right]^\frown \\
 & & \qquad \left[\begin{array}{l}e_7\text{:end}(e_7) \\ e_8\text{:end}(e_8) \\ e_9\text{:start}(e_9)^\frown\text{end}(e_9) \\ e_{10}\text{:start}(e_{10}) \\ e_{11}\text{:start}(e_{11})\end{array}\right]^\frown \left[\begin{array}{l}e_{10}\text{:end}(e_{10}) \\ e_{11}\text{:end}(e_{11}) \\ e_{12}\text{:end}(e_{12})\end{array}\right]
\end{array}\right]$$

The event labelled 'e_{12}' will be of type *NP* and thus satisfy the requirement of e_6. By carrying out the same procedure as before we will obtain a new event (labelled 'e_{13}') of type *VP* which will satisfy the requirement of 'e_3' which will allow us to add a new event (labelled 'e_{14}') of type *S* whose start is at the beginning of the string labelled 'e' and whose end is at the end of that string. The final chart type is given in (74).

(74)

$$\left[\begin{array}{lcl}
e_1 & : & \text{“Dudamel”} \\
e_2 & : & \text{Lex}_{\text{PropName}}(\text{“Dudamel”}, d) \mathbin{\dot\wedge} \left[\text{s-event:}\left[\text{e=}\Uparrow^2 e_1\text{:}\textit{Phon}\right]\ \right] \\
e_3 & : & \left[\begin{array}{l}\text{rule=}\textit{S} \longrightarrow \textit{NP VP} \mid \textit{NP}'(\textit{VP}')\text{:}(\textit{NP}^\frown \textit{VP} \rightarrow \textit{Type}) \\ \text{fnd=}\Uparrow e_2\text{:}\textit{Sign} \\ \text{req=}\textit{VP}\text{:}\textit{Type} \\ \text{e:required(req,rule)}\end{array}\right] \\
e_4 & : & \text{“is”} \\
e_5 & : & \text{Lex}_{\text{be}}(\text{“is”}) \mathbin{\dot\wedge} \left[\text{s-event:}\left[\text{e=}\Uparrow^2 e_4\text{:}\textit{Phon}\right]\ \right] \\
e_6 & : & \left[\begin{array}{l}\text{rule=}\textit{VP} \longrightarrow [_{\textit{V}}\ \text{“is”}]\ [_{\textit{NP}}\ [_{\textit{Det}}\ \text{“a”}]\ \textit{N}] \mid \textit{N}'\text{:} \\ \qquad (\textit{V} \mathbin{\dot\wedge} \left[\text{s-event:}\left[\text{e:“is”}\right]\right]^\frown \\ \qquad\quad \textit{NP} \mathbin{\dot\wedge} \left[\text{syn:}\left[\begin{array}{l}\text{daughters:}\textit{Det} \mathbin{\dot\wedge} \left[\text{s-event:}\left[\text{e:“a”}\right]\right] \\ \qquad\qquad\qquad ^\frown \textit{N} \mathbin{\dot\wedge} \left[\text{cnt:}\textit{Ppty}\right]\end{array}\right]\right] \\ \qquad\qquad \rightarrow \textit{Type}) \\ \text{fnd=}\Uparrow e_5\text{:}\textit{Sign} \\ \text{req=}\textit{NP}\text{:}\textit{Type} \\ \text{e:required(req,rule)}\end{array}\right] \\
e_7 & : & \text{“a”} \\
e_8 & : & \text{Lex}_{\text{IndefArt}}(\text{“a”}) \mathbin{\dot\wedge} \left[\text{s-event:}\left[\text{e=}\Uparrow^2 e_7\text{:}\textit{Phon}\right]\ \right] \\
e_9 & : & \left[\begin{array}{l}\text{rule=}\textit{NP} \longrightarrow \textit{Det N} \mid \textit{Det}'(\textit{N}')\text{:}(\textit{Det}^\frown \textit{N} \rightarrow \textit{Type}) \\ \text{fnd=}\Uparrow e_8\text{:}\textit{Sign} \\ \text{req=}\textit{N}\text{:}\textit{Type} \\ \text{e:required(req,rule)}\end{array}\right] \\
e_{10} & : & \text{“conductor”} \\
e_{11} & : & \text{Lex}_{\text{CommonNoun}}(\text{“conductor”, conductor}) \mathbin{\dot\wedge} \left[\text{s-event:}\left[\text{e=}\Uparrow^2 e_{10}\text{:}\textit{Phon}\right]\ \right] \\
e_{12} & : & e_9\text{.rule}(e_9\text{.fnd}^\frown e_{11}) \\
e_{13} & : & e_6\text{.rule}(e_6\text{.fnd}^\frown e_{12}) \\
e_{14} & : & e_3\text{.rule}(e_3\text{.fnd}^\frown e_{13}) \\
e & : & \left[\begin{array}{l}e_1\text{:start}(e_1) \\ e_2\text{:start}(e_2) \\ e_{14}\text{:start}(e_{14})\end{array}\right]^\frown \left[\begin{array}{l}e_1\text{:end}(e_1) \\ e_2\text{:end}(e_2) \\ e_3\text{:start}(e_3)^\frown\text{end}(e_3) \\ e_4\text{:start}(e_4) \\ e_5\text{:start}(e_5) \\ e_{13}\text{:start}(e_{13})\end{array}\right]^\frown \left[\begin{array}{l}e_4\text{:end}(e_4) \\ e_5\text{:end}(e_5) \\ e_6\text{:start}(e_6)^\frown\text{end}(e_6) \\ e_7\text{:start}(e_7) \\ e_8\text{:start}(e_8) \\ e_{12}\text{:start}(e_{12})\end{array}\right]^\frown \\
 & & \qquad \left[\begin{array}{l}e_7\text{:end}(e_7) \\ e_8\text{:end}(e_8) \\ e_9\text{:start}(e_9)^\frown\text{end}(e_9) \\ e_{10}\text{:start}(e_{10}) \\ e_{11}\text{:start}(e_{11})\end{array}\right]^\frown \left[\begin{array}{l}e_{10}\text{:end}(e_{10}) \\ e_{11}\text{:end}(e_{11}) \\ e_{12}\text{:end}(e_{12}) \\ e_{13}\text{:end}(e_{13}) \\ e_{14}\text{:end}(e_{14})\end{array}\right]
\end{array}\right]$$

@@ [Update functions to achieve this...]

Appendix A

Type theory with records

Unless otherwise stated this is the version of TTR presented in Cooper (2012b).

A.1 Underlying set theory

In previous statements of this system such as Cooper (2012b) we tacitly assumed a standard underlying set theory such as ZF (Zermelo-Fraenkel) with urelements (as formulated for example in Suppes, 1960). This is what we take to be the common or garden working set theory which is familiar from the core literature on formal semantics deriving from Montague's original work (Montague, 1974). When we introduced complex objects and types other than records and record types we were not explicit about exactly which structured set-theoretic object they represented. The reason for this was that, except in the case of records and record types, it did not seem important exactly how you code structured objects in the set theory and a detailed exposition would seem to provide another level of complication over and above an already complicated story.

In this version we will take advantage of the freedom provided by an appendix and spell out a set theoretic coding for all of our structured objects. We will use *labelled sets* to model our structured objects which is what we use for records and record types as well. We will assume that our set theory comes equipped with a set of *urelements* (objects which are not sets but which can be members of sets) which is partitioned into two countable subsets or *urelements proper* (intuitively "real atomic objects") and *labels* (intuitively "objects that are used to label real objects, either atomic or sets"). A *labelled set* is a set of ordered pairs whose first member is a label and whose second element is either an urelement proper or a set (possibly a labelled set), such that no more than one ordered pair can contain any particular label as its first member. This

means that a labelled set is the traditional set theoretic construction of an extensional function from a set of labels onto some set. Suppose that we have a set

$\{a, b, c, d\}$

and that $\ell_0, \ell_1, \ell_2, \ell_3$ are labels. Then examples of labelled sets would be

$\{\langle \ell_0, a\rangle, \langle \ell_1, b\rangle, \langle \ell_2, c\rangle, \langle \ell_3, d\rangle\}$

and

$\{\langle \ell_0, \{\langle \ell_0, a\rangle, \langle \ell_1, b\rangle\}\rangle, \langle \ell_2, c\rangle, \langle \ell_3, d\rangle\}$

Labelled sets where we identify particular distinguished labels will always give us enough structure to model the structured objects that we need and define operations on them as required by the type theory.

A.2 Basic types

A *system of basic types* is a pair:

$$\mathbf{TYPE}_B = \langle \mathbf{Type}, A\rangle$$

where:

1. **Type** is a non-empty set
2. A is a function whose domain is **Type**
3. for any $T \in$ **Type**, $A(T)$ is a set disjoint from **Type**
4. for any $T \in$ **Type**, $a :_{\mathbf{TYPE_B}} T$ iff $a \in A(T)$

A *modal system of basic types*[1] is a family of pairs:

$$\mathbf{TYPE}_{MB} = \langle \mathbf{Type}, A\rangle_{A \in \mathcal{A}}$$

[1]This definition was not present in Cooper (2012b).

where:

1. $\mathcal{A}$ is a set of functions with domain **Type**
2. for each $A \in \mathcal{A}$, $\langle$**Type**, $A\rangle$ is a system of basic types

This enables us to define some simple modal notions:

If **TYPE**$_{MB}$ = $\langle$**Type**, $A\rangle_{A\in\mathcal{A}}$ is a modal system of basic types, we shall use the notation **TYPE**$_{MB_A}$ (where $A \in \mathcal{A}$) to refer to that system of basic types in **TYPE**$_{MB}$ whose type assignment is A. Then:

1. for any $T_1, T_2 \in$ **Type**, T_1 *is (necessarily) equivalent to* T_2 *in* **TYPE**$_{MB}$, $T_1 \approx_{\mathbf{TYPE}_{MB}} T_2$, iff for all $A \in \mathcal{A}$, $\{a \mid a :_{\mathbf{TYPE}_{MB_A}} T_1\} = \{a \mid a :_{\mathbf{TYPE}_{MB_A}} T_2\}$
2. for any $T_1, T_2 \in$ **Type**, T_1 *is a subtype of* T_2 *in* **TYPE**$_{MB}$, $T_1 \sqsubseteq_{\mathbf{TYPE}_{MB}} T_2$, iff for all $A \in \mathcal{A}$, $\{a \mid a :_{\mathbf{TYPE}_{MB_A}} T_1\} \subseteq \{a \mid a :_{\mathbf{TYPE}_{MB_A}} T_2\}$
3. for any $T \in$ **Type**, T *is necessary in* **TYPE**$_{MB}$ iff for all $A \in \mathcal{A}$, $\{a \mid a :_{\mathbf{TYPE}_{MB_A}} T\} \neq \emptyset$
4. for any $T \in$ **Type**, T *is possible in* **TYPE**$_{MB}$ iff for some $A \in \mathcal{A}$, $\{a \mid a :_{\mathbf{TYPE}_{MB_A}} T\} \neq \emptyset$

A.3 Complex types

A.3.1 Predicates

We start by introducing the notion of a predicate signature.

A *predicate signature* is a triple

$$\langle \textbf{Pred}, \textbf{ArgIndices}, \mathit{Arity}\rangle$$

where:

1. **Pred** is a set (of predicates)

2. **ArgIndices** is a set (of indices for predicate arguments, normally types)
3. *Arity* is a function with domain **Pred** and range included in the set of finite sequences of members of **ArgIndices**.

A *polymorphic predicate signature* is a triple

$$\langle \textbf{Pred}, \textbf{ArgIndices}, \mathit{Arity} \rangle$$

where:

1. **Pred** is a set (of predicates)
2. **ArgIndices** is a set (of indices for predicate arguments, normally types)
3. *Arity* is a function with domain **Pred** and range included in the powerset of the set of finite sequences of members of **ArgIndices**.

A.3.2 Systems of complex types

A *system of complex types* is a quadruple:

$$\textbf{TYPE}_C = \langle \textbf{Type}, \textbf{BType}, \langle \textbf{PType}, \textbf{Pred}, \textbf{ArgIndices}, \mathit{Arity} \rangle, \langle A, F \rangle \rangle$$

where:

1. $\langle$**BType**, $A\rangle$ is a system of basic types
2. **BType**$\subseteq$**Type**
3. for any $T \in$ **Type**, if $a :_{\langle \mathbf{BType}, A\rangle} T$ then $a :_{\mathbf{TYPE_C}} T$
4. $\langle$**Pred**, **ArgIndices**, *Arity*$\rangle$ is a (polymorphic) predicate signature
5.[2] $P(a_1, \ldots a_n) \in$ **PType** iff $P \in$ **Pred**, $T_1 \in \mathbf{Type}, \ldots, T_n \in \mathbf{Type}$, *Arity*$(P)=\langle T_1, \ldots, T_n\rangle$ ($\langle T_1, \ldots, T_n\rangle \in$*Arity*$(P)$) and $a_1 :_{\mathbf{TYPE_C}} T_1, \ldots, a_n :_{\mathbf{TYPE_C}} T_n$

[2]This clause has been modified since Cooper (2012b) where it was a conditional rather than a biconditional.

6. **PType**$\subseteq$**Type**
7. for any $T \in$ **PType**, $F(T)$ is a set disjoint from **Type**
8. for any $T \in$ **PType**, $a :_{\mathbf{TYPE_C}} T$ iff $a \in F(T)$

We call the pair $\langle A, F\rangle$ in a complex system of types the *model* because of its similarity to first order models in providing values for the basic types and the ptypes constructed from predicates and arguments. It is this pair which connects the system of types to the non-type theoretical world of objects and situations.

In Cooper (2012b) we did not define exactly what object is represented by $P(a_1, \ldots a_n)$. Here we will specify it to be the labelled set

$\{\langle \mathrm{pred}, P\rangle, \langle \mathrm{arg}_1, a_1\rangle, \ldots, \langle \mathrm{arg}_n, a_n\rangle\}$

where 'pred', 'arg$_i$' are reserved labels (not used except as required here).

What are the objects which belong to these types? The intuition is that, for example,

e : run(a)

means that e is an event or situation where the individual a is running. There are two competing intuitions about what e could be. One is that it is a "part of the world", a non-set (urelement). That is, from the perspective of set theory and type theory it is an unstructured atom. The other intuition we have is that it is a structured object which contains a as a component and in which a running activity is going on which involves smaller events such as picking feet up off the ground, spending certain time in each step cycle with neither foot touching the ground and so on. We want to allow for both of these intuitions. That is, a witness for a ptype can be a non-set corresponding to our notion of an event of a certain type. Or it can be the kind of labelled set which we call a record. That is e does not only belong to the type 'run(a)' but also a record type which characterizes in more detail the structure of the event. We will argue in the text that both intuitions are important and that observers of the world shift between type theories where certain ptypes are regarded as types of non-sets and type theories where those ptypes are types of records.

A.4 Function types

A system of complex types **TYPE**$_C$ = $\langle$**Type**, **BType**, $\langle$**PType**, **Pred**, **ArgIndices**, *Arity*$\rangle$, $\langle A, F\rangle\rangle$ *has function types* if

1. for any $T_1, T_2 \in$ **Type**, $(T_1 \rightarrow T_2) \in$ **Type**

2. for any $T_1, T_2 \in$ **Type**, $f :_{\mathbf{TYPE_C}} (T_1 \rightarrow T_2)$ iff f is a function whose domain is $\{a \mid a :_{\mathbf{TYPE_C}} T_1\}$ and whose range is included in $\{a \mid a :_{\mathbf{TYPE_C}} T_2\}$

In Cooper (2012b) we did not specify exactly what object is represented by a function type $(T_1 \rightarrow T_2)$. Here we specify it to be the labelled set

$\{\langle \mathrm{dmn}, T_1\rangle, \langle \mathrm{rng}, T_2\rangle\}$

where 'dmn' ("domain") and 'rng' ("range") are reserved labels.

In Cooper (2012b) we also left it open exactly what kind of object a function is and assumed there was some theory of functions which would allow us to characterize them in terms of their domain and range. In a classical set theoretic setting where functions are modelled extensionally as sets of ordered pairs, little more needs to be said. Ideally, we want a notion of function that is more like a program or a procedure. That is, functions can be intensional in the sense that two distinct functions can correspond to the same set of ordered pairs. However, it seems that for the purposes at hand the standard extensional notion of function as a set of ordered pairs is sufficient and a lot more straightforward to handle than a more intensional notion given the set-theoretic basis on which we are building. There appears to be sufficient intensionality introduced by our notion of type and the of-type relation. For this reason, we will model functions here as sets of ordered pairs in the classical set-theoretic way. Ultimately, we suspect that a more computational and intensional notion of function should be substituted, but at this point it is unclear what consequences this might have for the rest of the system.

This choice of modelling functions as sets of ordered pairs means that $f :_{\mathbf{TYPE_C}} (T_1 \rightarrow T_2)$ iff $f \subseteq \{a \mid a :_{\mathbf{TYPE_C}} T_1\} \times \{a \mid a :_{\mathbf{TYPE_C}} T_2\}$ such that if $b \in \{a \mid a :_{\mathbf{TYPE_C}} T_1\}$ then there is exactly one c, such that $\langle b, c\rangle \in f$. We shall say that in this case the result of applying the function f to b, in symbols, $f(b)$, is c.

We introduce a notation for functions which is borrowed from the λ-calculus as used by Montague (1973). Let $O[v]$ be the notation for some object of our type theory which uses the variable v and let T be a type. Then the function

$\lambda v : T \,.\, O[v]$

is to be the function

$\{\langle v, O[v]\rangle \mid v : T\}$

(Here we suppress the subscript $\mathbf{TYPE_C}$ on the ':'.) For example, the function

λv:*Ind* . run(v)

is the set of ordered pairs

$\{\langle v, \mathrm{run}(v)\rangle \mid v : Ind\}$

Recall that 'run(v)' is itself a representation for the labelled set

$\{\langle \mathrm{pred}, \mathrm{run}\rangle, \langle \mathrm{arg}_1, v\rangle\}$

Note that if f is the function λv:*Ind* . run(v) and a:*Ind* then $f(a)$ (the result of applying f to a) is 'run(a)'. Our definition of function-argument application guarantees what is called β-equivalence in the λ-calculus. When we discuss record types as arguments to functions we will need to introduce one slight complication to our notion of function application. We will introduce that complication when we discuss record types.

A.5 List types

List types were not included in Cooper (2012b).

A system of complex types **TYPE**$_C$ = ⟨**Type**, **BType**, ⟨**PType**, **Pred**, **ArgIndices**, *Arity*⟩, ⟨A, F⟩⟩ *has list types* if

1. for any $T \in$ **Type**, $[T] \in$ **Type**
2. for any $T \in$ **Type**,
   (a) $a \mid L :_{\mathbf{TYPE_C}} [T]$ iff $a :_{\mathbf{TYPE_C}} T$ and $L :_{\mathbf{TYPE_C}} [T]$
   (b) $[\,] :_{\mathbf{TYPE_C}} [T]$

A system of complex types **TYPE**$_C$ = ⟨**Type**, **BType**, ⟨**PType**, **Pred**, **ArgIndices**, *Arity*⟩, ⟨A, F⟩⟩ *has non-empty list types* if

1. for any $T \in$ **Type**, $_{ne}[T] \in$ **Type**
2. for any $T \in$ **Type**,
   (a) $a \mid L :_{\mathbf{TYPE_C}} {}_{ne}[T]$ iff $a :_{\mathbf{TYPE_C}} T$ and $L :_{\mathbf{TYPE_C}} {}_{ne}[T]$
   (b) $[a] :_{\mathbf{TYPE_C}} {}_{ne}[T]$ iff $a : T$

If $a \mid L :_{\mathbf{TYPE_C}} {}_{ne}[T]$ for some system of complex types $\mathbf{TYPE}_C$ and type T, then we use fst($a \mid L$) to refer to a and rst($a \mid L$) to refer to L.

In contrast to Cooper (2012b) we here make it explcit that $[T]$ represents $\{\langle \mathrm{lst}, T\rangle\}$ and ${}_{ne}[T]$ represents $\{\langle \mathrm{nelst}, T\rangle\}$ where 'lst' and 'nelst' are reserved labels.

Lists are a common data structure used in computer science but they are not normally defined in basic set theory, although it is straightforward to define them in terms of sets. In Cooper (2012b) we did not specify an encoding of lists in terms of sets. Here we will use an encoding with labelled sets using the reserved labels 'fst' and 'rst' for the first member of the list and the remainder ("rest") of the list respectively. We let the empty list, [], be the empty set, $\emptyset$.[3] If L is a list then $a \mid L$ is to be the labelled set $\{\langle \mathrm{fst}, a\rangle, \langle \mathrm{rst}, L\rangle\}$.

A.6 Set types

Set types were not included in Cooper (2012b).

A system of complex types $\mathbf{TYPE}_C$ = $\langle$**Type**, **BType**, $\langle$**PType**, **Pred**, **ArgIndices**, *Arity*$\rangle$, $\langle A, F\rangle\rangle$ *has set types* if

1. for any $T \in$ **Type**, $\{T\} \in$ **Type**
2. for any $T \in$ **Type**, $A :_{\mathbf{TYPE_C}} \{T\}$ iff A is a set and for all $a \in A$, $a :_{\mathbf{TYPE_C}} T$

We let $\{T\}$ represent the labelled set $\{\langle \mathrm{set}, T\rangle\}$ where 'set' is a reserved labelled.

A.7 Join types

A system of complex types $\mathbf{TYPE}_C$ = $\langle$**Type**, **BType**, $\langle$**PType**, **Pred**, **ArgIndices**, *Arity*$\rangle$, $\langle A, F\rangle\rangle$ *has join types* if

1. for any $T_1, T_2 \in \mathbf{Type}$, $(T_1 \vee T_2) \in \mathbf{Type}$
2. for any $T_1, T_2 \in \mathbf{Type}$, $a :_{\mathbf{TYPE_C}} (T_1 \vee T_2)$ iff $a :_{\mathbf{TYPE_C}} T_1$ or $a :_{\mathbf{TYPE_C}} T_2$

[3] If it is important to distinguish the empty list from the empty set we could use an additional reserved label, e.g. 'lst', and have the empty list be the labelled set $\{\langle \mathrm{lst}, \emptyset\rangle\}$.

Here, but not in Cooper (2012b), we specify that $(T_1 \vee T_2)$ represents the labelled set $\{\langle \mathrm{disj}_1, T_1\rangle, \langle \mathrm{disj}_2, T_2\rangle\}$ where 'disj$_1$' and 'disj$_2$' are reserved labels ("disjunct").

A.8 Meet types

A system of complex types **TYPE**$_C$ = $\langle$**Type**, **BType**, $\langle$**PType**, **Pred**, **ArgIndices**, *Arity*$\rangle$, $\langle A, F\rangle\rangle$ *has meet types* if

1. for any $T_1, T_2 \in \mathbf{Type}$, $(T_1 \wedge T_2) \in \mathbf{Type}$
2. for any $T_1, T_2 \in \mathbf{Type}$, $a :_{\mathbf{TYPE_C}} (T_1 \wedge T_2)$ iff $a :_{\mathbf{TYPE_C}} T_1$ and $a :_{\mathbf{TYPE_C}} T_2$

Here, but not in Cooper (2012b), we specify that $(T_1 \wedge T_2)$ represents the labelled set $\{\langle \mathrm{conj}_1, T_1\rangle, \langle \mathrm{conj}_2, T_2\rangle\}$ where 'conj$_1$' and 'conj$_2$' are reserved labels ("conjunct").

A.9 Models and modal systems of types

A modal system of complex types provides a collection of models, $\mathcal{M}$, so that we can talk about properties of the whole collection of type assignments provided by the various models $M \in \mathcal{M}$.

A *modal system of complex types based on* $\mathcal{M}$ is a family of quadruples[4]:

$$\mathbf{TYPE}_{MC} = \langle \mathbf{Type}_M, \mathbf{BType}, \langle \mathbf{PType}_M, \mathbf{Pred}, \mathbf{ArgIndices}, \mathit{Arity}\rangle, M\rangle_{M\in\mathcal{M}}$$

where for each $M \in \mathcal{M}$, $\langle \mathbf{Type}_M, \mathbf{BType}, \langle \mathbf{PType}_M, \mathbf{Pred}, \mathbf{ArgIndices}, \mathit{Arity}\rangle, M\rangle$ is a system of complex types.

This enables us to define modal notions:

If **TYPE**$_{MC}$ = $\langle \mathbf{Type}_M, \mathbf{BType}, \langle \mathbf{PType}_M, \mathbf{Pred}, \mathbf{ArgIndices}, \mathit{Arity}\rangle, M\rangle_{M\in\mathcal{M}}$ is a modal system of complex types based on $\mathcal{M}$, we shall use the notation **TYPE**$_{MC_M}$ (where $M \in \mathcal{M}$) to refer to that system of complex types in **TYPE**$_{MC}$ whose model is M. Let **Type**$_{MC_{restr}}$ be $\bigcap_{M\in\mathcal{M}}$ **Type**$_M$,

[4]This definition has been modified since Cooper (2012b) to make **PType** and **Type** be relativized to the model M.

the "restrictive" set of types which occur in all possibilities, and $\mathbf{Type}_{MC_{incl}}$ be $\bigcup_{M \in \mathcal{M}} \mathbf{Type}_M$, the "inclusive" set of types which occur in at least one possibility. Then we can define modal notions either restrictively or inclusively (indicated by the subscripts r and i respectively):

restrictive modal notions

1. for any $T_1, T_2 \in \mathbf{Type}_{MC_{restr}}$, T_1 *is (necessarily) equivalent$_r$ to* T_2 *in* $\mathbf{TYPE}_{MC}$, $T_1 \approx_{\mathbf{TYPE}_{MC}} T_2$, iff for all $M \in \mathcal{M}$, $\{a \mid a :_{\mathbf{TYPE}_{MC_M}} T_1\} = \{a \mid a :_{\mathbf{TYPE}_{MC_M}} T_2\}$

2. for any $T_1, T_2 \in \mathbf{Type}_{MC_{restr}}$, T_1 *is a subtype$_r$ of* T_2 *in* $\mathbf{TYPE}_{MC}$, $T_1 \sqsubseteq_{\mathbf{TYPE}_{MC}} T_2$, iff for all $M \in \mathcal{M}$, $\{a \mid a :_{\mathbf{TYPE}_{MC_M}} T_1\} \subseteq \{a \mid a :_{\mathbf{TYPE}_{MC_M}} T_2\}$

3. for any $T \in \mathbf{Type}_{MC_{restr}}$, T *is necessary$_r$ in* $\mathbf{TYPE}_{MC}$ iff for all $M \in \mathcal{M}$, $\{a \mid a :_{\mathbf{TYPE}_{MC_M}} T\} \neq \emptyset$

4. for any $T \in \mathbf{Type}_{MC_{restr}}$, T *is possible$_r$ in* $\mathbf{TYPE}_{MC}$ iff for some $M \in \mathcal{M}$, $\{a \mid a :_{\mathbf{TYPE}_{MC_M}} T\} \neq \emptyset$

inclusive modal notions

1. for any $T_1, T_2 \in \mathbf{Type}_{MC_{incl}}$, T_1 *is (necessarily) equivalent$_i$ to* T_2 *in* $\mathbf{TYPE}_{MC}$, $T_1 \approx_{\mathbf{TYPE}_{MC}} T_2$, iff for all $M \in \mathcal{M}$, if T_1 and T_2 are members of $\mathbf{Type}_M$, then $\{a \mid a :_{\mathbf{TYPE}_{MC_M}} T_1\} = \{a \mid a :_{\mathbf{TYPE}_{MC_M}} T_2\}$

2. for any $T_1, T_2 \in \mathbf{Type}_{MC_{incl}}$, T_1 *is a subtype$_i$ of* T_2 *in* $\mathbf{TYPE}_{MC}$, $T_1 \sqsubseteq_{\mathbf{TYPE}_{MC}} T_2$, iff for all $M \in \mathcal{M}$, if T_1 and T_2 are members of $\mathbf{Type}_M$, then $\{a \mid a :_{\mathbf{TYPE}_{MC_M}} T_1\} \subseteq \{a \mid a :_{\mathbf{TYPE}_{MC_M}} T_2\}$

3. for any $T \in \mathbf{Type}_{MC_{incl}}$, T *is necessary$_i$ in* $\mathbf{TYPE}_{MC}$ iff for all $M \in \mathcal{M}$, if $T \in \mathbf{Type}_M$, then $\{a \mid a :_{\mathbf{TYPE}_{MC_M}} T\} \neq \emptyset$

4. for any $T \in \mathbf{Type}_{MC_{incl}}$, T *is possible$_i$ in* $\mathbf{TYPE}_{MC}$ iff for some $M \in \mathcal{M}$, if $T \in \mathbf{Type}_M$, then $\{a \mid a :_{\mathbf{TYPE}_{MC_M}} T\} \neq \emptyset$

It is easy to see that if any of the restrictive definitions holds for given types in a particular system then the corresponding inclusive definition will also hold for those types in that system.

A.10 The type *Type* and stratification

An *intensional system of complex types* is a family of quadruples indexed by the natural numbers:

$$\textbf{TYPE}_{IC} = \langle \textbf{Type}^n, \textbf{BType}, \langle \textbf{PType}^n, \textbf{Pred}, \textbf{ArgIndices}, \mathit{Arity}\rangle, \langle A, F^n\rangle\rangle_{n\in Nat}$$

where (using $\mathbf{TYPE}_{IC_n}$ to refer to the quadruple indexed by n):

1. for each n,$\langle$**Type**^{n}, **BType**, $\langle$**PType**^{n}, **Pred**, **ArgIndices**, *Arity*$\rangle$, $\langle A, F^n\rangle\rangle$ is a system of complex types
2. for each n, $\mathbf{Type}^n \subseteq \mathbf{Type}^{n+1}$ and $\mathbf{PType}^n \subseteq \mathbf{PType}^{n+1}$
3. for each n, if $T \in \mathbf{PType}^n$ and $p \in F^n(T)$ then $p \in F^{n+1}(T)$
4. for each $n > 0$, $\mathit{Type}^n \in \mathbf{Type}^n$
5. for each $n > 0$, $T :_{\mathbf{TYPE}_{IC_n}} \mathit{Type}^n$ iff $T \in \mathbf{Type}^{n-1}$

Here, but not in Cooper (2012b), we make explicit that *Type* is a distinguished urelement and that *Type*^{n} represents the labelled set $\{\langle \mathrm{ord}, n\rangle, \langle \mathrm{typ}, \mathit{Type}\rangle\}$ where 'ord' and 'typ' are reserved labels ("order", "type").

An intensional system of complex types $\mathbf{TYPE}_{IC}$,

$$\textbf{TYPE}_{IC} = \langle \textbf{Type}^n, \textbf{BType}, \langle \textbf{PType}^n, \textbf{Pred}, \textbf{ArgIndices}, \mathit{Arity}\rangle, \langle A, F^n\rangle\rangle_{n\in Nat}$$

has dependent function types if

1. for any $n > 0$, $T \in \mathbf{Type}^n$ and $\mathcal{F} :_{\mathbf{TYPE}_{IC_n}} (T \rightarrow \mathit{Type}^n)$, $((a : T) \rightarrow \mathcal{F}(a)) \in \mathbf{Type}^n$
2. for each $n > 0$, $f :_{\mathbf{TYPE}_{IC_n}} ((a : T) \rightarrow \mathcal{F}(a))$ iff f is a function whose domain is $\{a \mid a :_{\mathbf{TYPE}_{IC_n}} T\}$ and such that for any a in the domain of f, $f(a) :_{\mathbf{TYPE}_{IC_n}} \mathcal{F}(a)$.

We might say that on this view dependent function types are "semi-intensional" in that they depend on there being a type of types for their definition but they do not introduce types as arguments to predicates and do not involve the definition of orders of types in terms of the types of the next lower order.

Here, in contrast to Cooper (2012b), we make explicit that $((a : T) \rightarrow \mathcal{F}(a))$ represents the labelled set $\{\langle \mathrm{dmn}, T\rangle, \langle \mathrm{deprng}, \mathcal{F}\rangle\}$ where 'dmn' as before for function types is a reserved label corresponding to "domain" and 'deprng' is a reserved label corresponding to "dependent range".

Putting the definition of a modal type system and an intensional type system together we obtain:[5]

An *intensional modal system of complex types based on* $\mathfrak{M}$ is a family, indexed by the natural numbers, of families of quadruples indexed by members of $\mathfrak{M}$:

$$\textbf{TYPE}_{IMC} = \langle \textbf{Type}^n, \textbf{BType}, \langle \textbf{PType}^n, \textbf{Pred}, \textbf{ArgIndices}, \mathit{Arity}\rangle, \mathcal{M}_n\rangle_{\mathcal{M}\in\mathfrak{M}, n\in Nat}$$

where:

1. for each n, $\langle \textbf{Type}^n, \textbf{BType}, \langle \textbf{PType}^n, \textbf{Pred}, \textbf{ArgIndices}, \mathit{Arity}\rangle, \mathcal{M}_n\rangle_{\mathcal{M}\in\mathfrak{M}}$ is a modal system of complex types based on $\{\mathcal{M}_n \mid \mathcal{M} \in \mathfrak{M}\}$
2. for each $\mathcal{M} \in \mathfrak{M}$, $\langle \textbf{Type}^n, \textbf{BType}, \langle \textbf{PType}^n, \textbf{Pred}, \textbf{ArgIndices}, \mathit{Arity}\rangle, \mathcal{M}_n\rangle_{n\in Nat}$ is an intensional system of complex types

A.11 Record types

In this section we will define what it means for a system of complex types to have record types. The objects of record types, that is, records, are themselves structured mathematical objects of a particular kind and we will start by characterizing them.

A *record* is a finite set of ordered pairs (called *fields*) which is the graph of a function. If r is a record and $\langle \ell, v\rangle$ is a field in r we call ℓ a *label* and v a *value* in r and we use $r.\ell$ to denote v. $r.\ell$ is called a *path* in r. This means that a record is a labelled set as introduced in Appendix A.1.

We will use a tabular format to represent records. A record $\{\langle \ell_1, v_1\rangle, \ldots, \langle \ell_n, v_n\rangle\}$ is displayed as

$$\begin{bmatrix} \ell_1 & = & v_1 \\ \cdots & & \\ \ell_n & = & v_n \end{bmatrix}$$

[5]This explicit definition was not present in Cooper (2012b).

A value may itself be a record and paths may extend into embedded records. A record which contains records as values is called a *complex record* and otherwise a record is *simple*. Values which are not records are called *leaves*. Consider a record r

$$\left[\begin{array}{lcl} f & = & \left[\begin{array}{lcl} f & = & \left[\begin{array}{lcl} \mathit{ff} & = & a \\ gg & = & b \end{array}\right] \\ g & = & c \end{array}\right] \\ g & = & \left[\begin{array}{lcl} h & = & \left[\begin{array}{lcl} g & = & a \\ h & = & d \end{array}\right] \end{array}\right] \end{array}\right]$$

Among the paths in r are $r.f$, $r.g.h$ and $r.f.f.\mathit{ff}$ which denote, respectively,

$$\left[\begin{array}{lcl} f & = & \left[\begin{array}{lcl} \mathit{ff} & = & a \\ gg & = & b \end{array}\right] \\ g & = & c \end{array}\right]$$

$$\left[\begin{array}{lcl} g & = & a \\ h & = & d \end{array}\right]$$

and a. We will make a distinction between *absolute paths*, such as those we have already mentioned, which consist of a record followed by a series of labels connected by dots and *relative paths* which are just a series of labels connected by dots, e.g. $g.h$. Relative paths are useful when we wish to refer to similar paths in different records. We will use *path* to refer to either absolute or relative paths when it is clear from the context which is meant. The set of leaves of r, also known as its *extension* (those objects other than labels which it contains), is $\{a, b, c, d\}$. The bag (or multiset) of leaves of r, also known as its *multiset extension*, is $\{a, a, b, c, d\}$. A record may be regarded as a way of labelling and structuring its extension. Two records are *(multiset) extensionally equivalent* if they have the same (multiset) extension. Two important, though trivial, facts about records are:

Flattening. For any record r, there is a multiset extensionally equivalent simple record. We can define an operation of flattening on records which will always produce an equivalent simple record. In the case of our example, the result of flattening is

$$\left[\begin{array}{lcl} f.f.\mathit{ff} & = & a \\ f.f.gg & = & b \\ f.g & = & c \\ g.h.g & = & a \\ g.h.h & = & d \end{array}\right]$$

assuming the flattening operation uses paths from the original record in a rather obvious way to create unique labels for the new record.

Relabelling. For any record r, if $\pi_1.\ell.\pi_2$ is a path π in r, and $\pi_1.\ell'.\pi_2{}'$ is *not* a path in r (for any $\pi_2{}'$), then substituting ℓ' for the occurrence of ℓ in π results in a record which is multiset equivalent to r. We could, for example, substitute k for the second occurrence of g in the path $g.h.g$ in our example record.

$$\left[\begin{array}{lcl} f & = & \left[\begin{array}{lcl} f & = & \left[\begin{array}{lcl} ff & = & a \\ gg & = & b \end{array}\right] \\ g & = & c \end{array}\right] \\ g & = & \left[\begin{array}{lcl} h & = & \left[\begin{array}{lcl} k & = & a \\ h & = & d \end{array}\right] \end{array}\right] \end{array}\right]$$

A record *type* is a labelled set where the objects labelled are types or, in some cases, certain kinds of mathematical objects which can be used to construct types.

A record r is *well-typed* with respect to a system of types **TYPE** with set of types **Type** and a set of labels L iff for each field $\langle \ell, a \rangle \in r$, $\ell \in L$ and either $a :_{\mathbf{TYPE}} T$ for some $T \in \mathbf{Type}$ or a is itself a record which is well-typed with respect to **TYPE** and L.

A system of complex types $\mathbf{TYPE}_C$ = $\langle$**Type**, **BType**, $\langle$**PType**, **Pred**, **ArgIndices**, *Arity*$\rangle$, $\langle A, F \rangle\rangle$ *has record types based on* $\langle L, \mathbf{RType} \rangle$, where L is a countably infinite set (of labels) and **RType** $\subseteq$ **Type**, where **RType** is defined by:

1. $Rec \in \mathbf{RType}$
2. $r :_{\mathbf{TYPE}_C} Rec$ iff r is a well-typed record with respect to $\mathbf{TYPE}_C$ and L.
3. if $\ell \in L$ and $T \in \mathbf{Type}$, then $\{\langle \ell, T \rangle\} \in \mathbf{RType}$.
4. $r :_{\mathbf{TYPE}_C} \{\langle \ell, T \rangle\}$ iff $r :_{\mathbf{TYPE}_C} Rec$, $\langle \ell, a \rangle \in r$ and $a :_{\mathbf{TYPE}_C} T$.
5. if $R \in \mathbf{RType}$, $\ell \in L$, ℓ does not occur as a label in R (i.e. there is no field $\langle \ell', T' \rangle$ in R such that $\ell' = \ell$), then $R \cup \{\langle \ell, T \rangle\} \in \mathbf{RType}$.
6. $r :_{\mathbf{TYPE}_C} R \cup \{\langle \ell, T \rangle\}$ iff $r :_{\mathbf{TYPE}_C} R$, $\langle \ell, a \rangle \in r$ and $a :_{\mathbf{TYPE}_C} T$.

This gives us non-dependent record types in a system of complex types. We can extend this to intensional systems of complex types (with stratification).

An *intensional system of complex types* $\mathbf{TYPE}_{IC}$ = $\langle$**Type**^{n}, **BType**, $\langle$**PType**^{n}, **Pred**, **ArgIndices**, *Arity*$\rangle$, $\langle A, F^n \rangle\rangle_{n \in Nat}$ *has record types based on* $\langle L, \mathbf{RType}^n \rangle_{n \in Nat}$ if for each n, $\langle$**Type**^{n}, **BType**, $\langle$**PType**^{n}, **Pred**, **ArgIndices**, *Arity*$\rangle$, $\langle A, F^n \rangle\rangle$ has record types based on $\langle L, \mathbf{RType}^n \rangle$ and

1. for each n, **RType**$^n \subseteq$ **RType**$^{n+1}$
2. for each $n > 0$, *RecType*$^n \in$ **RType**^{n}
3. for each $n > 0$, $T :_{\mathbf{TYPE}_{IC_n}} RecType^n$ iff $T \in \mathbf{RType}^{n-1}$

Here, but not in Cooper (2012b), wwe make explicit that *RecType* is treated in a similar manner to *Type*, that is, it is a distinguished urelement and *RecType*^{n} represents the labelled set $\{\langle \mathrm{ord}, n\rangle, \langle \mathrm{typ}, RecType\rangle\}$ where 'ord' and 'typ' are reserved labels ("order", "type").

Intensional type systems may in addition contain *dependent* record types.

An *intensional system of complex types* **TYPE**$_{IC}$ = $\langle$**Type**^{n}, **BType**, $\langle$**PType**^{n}, **Pred**, **ArgIndices**, *Arity*$\rangle$, $\langle A, F^n\rangle\rangle_{n\in Nat}$ *has dependent record types based on* $\langle L, \mathbf{RType}^n\rangle_{n\in Nat}$, if it has records types based on $\langle L, \mathbf{RType}^n\rangle_{n\in Nat}$ and for each $n > 0$

1. if R is a member of **RType**^{n}, $\ell \in L$ not occurring as a label in R, $T_1, \ldots, T_m \in \mathbf{Type}^n$, $R.\pi_1, \ldots, R.\pi_m$ are paths in R and $\mathcal{F}$ is a function of type $((a_1 : T_1) \rightarrow \ldots \rightarrow ((a_m : T_m) \rightarrow Type^n)\ldots)$, then $R \cup \{\langle \ell, \langle \mathcal{F}, \langle \pi_1, \ldots, \pi_m\rangle\rangle\rangle\} \in \mathbf{RType}^n$.
2. $r :_{\mathbf{TYPE}_{IC_n}} R \cup \{\langle \ell, \langle \mathcal{F}, \langle \pi_1, \ldots, \pi_m\rangle\rangle\rangle\}$ iff $r :_{\mathbf{TYPE}_{IC_n}} R$, $\langle \ell, a\rangle$ is a field in r, $r.\pi_1 :_{\mathbf{TYPE}_{IC_n}} T_1, \ldots, r.\pi_m :_{\mathbf{TYPE}_{IC_n}} T_m$ and $a :_{\mathbf{TYPE}_{IC_n}} \mathcal{F}(r.\pi_1, \ldots, r.\pi_m)$.

We represent a record type $\{\langle \ell_1, T_1\rangle, \ldots, \langle \ell_n, T_n\rangle\}$ graphically as

$$\begin{bmatrix} \ell_1 & : & T_1 \\ \cdots & & \\ \ell_n & : & T_n \end{bmatrix}$$

In the case of dependent record types we sometimes use a convenient notation representing e.g.

$$\langle \lambda u \lambda v\ \mathrm{love}(u, v), \langle \pi_1, \pi_2\rangle\rangle$$

as

$$\mathrm{love}(\pi_1, \pi_2)$$

Our systems now allow both function types and dependent record types and allow dependent record types to be arguments to functions. We have to be careful when considering what the result of applying a function to a dependent record type should be. Consider the following simple example:

$$\lambda v_0 : \mathit{RecType}\,(\left[\mathrm{c}_0{:}v_0\right])$$

What should be the result of applying this function to the record type

$$\left[\begin{array}{lcl} \mathrm{x} & : & \mathit{Ind} \\ \mathrm{c}_1 & : & \langle \lambda v_1 : \mathit{Ind}(\mathrm{dog}(v_1)), \langle \mathrm{x} \rangle \rangle \end{array}\right]$$

Given normal assumptions about function application the result would be

$$\left[\begin{array}{lcl} \mathrm{c}_0 & : & \left[\begin{array}{lcl} \mathrm{x} & : & \mathit{Ind} \\ \mathrm{c}_1 & : & \langle \lambda v_1 : \mathit{Ind}\,(\mathrm{dog}(v_1)), \langle \mathrm{x} \rangle \rangle \end{array}\right] \end{array}\right] \text{(incorrect!)}$$

but this would be incorrect. In fact it is not a well-formed record type since x is not a path in it. Instead the result should be

$$\left[\begin{array}{lcl} \mathrm{c}_0 & : & \left[\begin{array}{lcl} \mathrm{x} & : & \mathit{Ind} \\ \mathrm{c}_1 & : & \langle \lambda v_1 : \mathit{Ind}\,(\mathrm{dog}(v_1)), \langle \mathrm{c}_0.\mathrm{x} \rangle \rangle \end{array}\right] \end{array}\right]$$

where the path from the top of the record type is specified. However, in the abbreviatory notation we write just 'x' when the label is used as an argument and interpret this as the path from the top of the record type to the field labelled 'x' in the local record type. Thus we will write

$$\left[\begin{array}{lcl} \mathrm{x} & : & \mathit{Ind} \\ \mathrm{c}_1 & : & \mathrm{dog(x)} \end{array}\right]$$

(where the 'x' in 'dog(x)' signifies the path consisting of just the single label 'x') and

$$\left[\begin{array}{lcl} \mathrm{c}_0 & : & \left[\begin{array}{lcl} \mathrm{x} & : & \mathit{Ind} \\ \mathrm{c}_1 & : & \mathrm{dog(x)} \end{array}\right] \end{array}\right]$$

(where the 'x' in 'dog(x)' signifies the path from the top of the record type down to 'x' in the local record type, that is, 'c_0.x').[6]

Note that this adjustment of paths is only required when a record type is being substituted into a position that lies on a path within a resulting record type. It will not, for example, apply in a case where a record type is to be substituted for an argument to a predicate such as when applying the function

[6]This convention of representing the path from the top of the record type to the "local" field by the final label on the path is new since Cooper (2012b).

$$\lambda v_0 : \textit{RecType} \left(\left[\mathrm{c}_0{:}\mathrm{appear}(v_0)\right]\right)$$

to

$$\left[\begin{array}{lcl} \mathrm{x} & : & \textit{Ind} \\ \mathrm{c}_1 & : & \langle \lambda v : \textit{Ind}\ (\mathrm{dog}(v)), \langle \mathrm{x} \rangle \rangle \\ \mathrm{c}_2 & : & \langle \lambda v : \textit{Ind}\ (\mathrm{approach}(v)), \langle \mathrm{x} \rangle \rangle \end{array}\right]$$

where the position of v_0 is in an "intensional context", that is, as the argument to a predicate and there is no path to this position in the record type resulting from applying the function. Here the result of the application is

$$\left[\begin{array}{lcl} \mathrm{c}_0 & : & \mathrm{appear}\left(\left[\begin{array}{lcl} \mathrm{x} & : & \textit{Ind} \\ \mathrm{c}_1 & : & \langle \lambda v : \textit{Ind}\ (\mathrm{dog}(v)), \langle \mathrm{x} \rangle \rangle \\ \mathrm{c}_2 & : & \langle \lambda v : \textit{Ind}\ (\mathrm{approach}(v)), \langle \mathrm{x} \rangle \rangle \end{array}\right]\right) \end{array}\right]$$

with no adjustment necessary to the paths representing the dependencies.[7] (Note that 'c_0.x' is not a path in this record type.)

Suppose that we wish to represent a type which requires that there is some dog such that it appears to be approaching (that is a *de re* reading). In the abbreviatory notation we might be tempted to write

$$\left[\begin{array}{lcl} \mathrm{x} & : & \textit{Ind} \\ \mathrm{c}_1 & : & \mathrm{dog(x)} \\ \mathrm{c}_0 & : & \mathrm{appear}(\left[\begin{array}{lcl} \mathrm{c}_2 & : & \mathrm{approach(x)} \end{array}\right]) \end{array}\right] \text{(incorrect!)}$$

corresponding to

$$\left[\begin{array}{lcl} \mathrm{x} & : & \textit{Ind} \\ \mathrm{c}_1 & : & \langle \lambda v{:}\textit{Ind}\ (\mathrm{dog}(v)), \langle \mathrm{x} \rangle \rangle \\ \mathrm{c}_0 & : & \mathrm{appear}(\left[\begin{array}{lcl} \mathrm{c}_2 & : & \langle \lambda v{:}\textit{Ind}\ (\mathrm{approach}(v)), \langle \mathrm{x} \rangle \rangle \end{array}\right]) \end{array}\right] \text{(incorrect!)}$$

This is, however, in correct since it refers to a path 'x' in the type which is the argument to 'appear' which does not exist. Instead we need to refer to the path 'x' in the record type containing the field labelled 'c_0':

[7] This record corresponds to the interpretation of *it appears that a dog is approaching*.

$$\left[\begin{array}{lcl} \text{x} & : & \textit{Ind} \\ \text{c}_1 & : & \langle \lambda v{:}\textit{Ind}\ (\text{dog}(v)), \langle \text{x} \rangle \rangle \\ \text{c}_0 & : & \langle \lambda v{:}\textit{Ind}\ (\text{appear}(\left[\begin{array}{lcl} \text{c}_2 & : & \text{approach}(v) \end{array}\right])), \langle \text{x} \rangle \rangle \end{array}\right]$$

In the abbreviatory notation we will use '⇑' to indicate that the path referred to is in the "next higher" record type[8]:

$$\left[\begin{array}{lcl} \text{x} & : & \textit{Ind} \\ \text{c}_1 & : & \text{dog(x)} \\ \text{c}_0 & : & \text{appear}(\left[\begin{array}{lcl} \text{c}_2 & : & \text{approach}(\Uparrow \text{x}) \end{array}\right]) \end{array}\right]$$

These matters arise as a result of our choice of using paths to represent dependencies in record types (rather than, for example, introducing additional unique identifiers to keep track of the positions within a record type as has been suggested by Thierry Coquand). It seems like a matter of implementation rather than a matter of substance and it is straightforward to define a path-aware notion of substitution which can be used in the definition of what it means to apply a TTR function to an argument. If f is a function represented by $\lambda v : T(\phi)$ and α is the representation of an object of type T, then the result of applying f to α, $f(\alpha)$, is represented by Subst(α,v,ϕ,$\emptyset$), that is, the result of substituting α for v in ϕ with respect to the empty path where for arbitrary α, v, ϕ, π, Subst(α,v,ϕ,π) is defined as

1. extend-paths(α,π), if ϕ is v
2. ϕ, if ϕ is of the form $\lambda v : T(\zeta)$, for some T and ζ (i.e. don't do any substitution if v is bound within ϕ)
3. $\lambda u : T$(Subst(α,v,ζ,π)), if ϕ is of the form $\lambda u : T(\zeta)$ and u is not v.
4. $\left[\begin{array}{lcl} \ell_1 & : & \text{Subst}(\alpha,v,T_1,\pi.\ell_1) \\ \ldots & & \\ \ell_n & : & \text{Subst}(\alpha,v,T_n,\pi.\ell_n) \end{array}\right]$, if ϕ is $\left[\begin{array}{lcl} \ell_1 & : & T_1 \\ \ldots & & \\ \ell_n & : & T_n \end{array}\right]$
5. P(Subst(α,v,β_1,π),...,Subst(α,v,β_n,π)), if α is $P(\beta_1, \ldots, \beta_n)$ for some predicate P
6. ϕ otherwise

extend-paths(α,π) is

1. $\langle f, \langle \pi.\pi_1, \ldots, \pi.\pi_n \rangle \rangle$, if α is $\langle f, \langle \pi_1, \ldots, \pi_n \rangle \rangle$

[8]This notation is new since Cooper (2012b).

2. $\begin{bmatrix} \ell_1 & : & \text{extend-paths}(T_1, \pi) \\ \dots & & \\ \ell_n & : & \text{extend-paths}(T_n, \pi) \end{bmatrix}$ if α is $\begin{bmatrix} \ell_1 & : & T_1 \\ \dots & & \\ \ell_n & : & T_n \end{bmatrix}$

3. P(extend-paths(β_1, π),. . . ,extend-paths(β_n, π)), if α is $P(\beta_1, \ldots, \beta_n)$ for some predicate P

4. α, otherwise

A.12 Merges of record types

If T_1 and T_2 are record types then there will always be a record type (not a meet) T_3 which is necessarily equivalent to $T_1 \wedge T_2$. Let us consider some examples:

$$\big[\text{f}{:}T_1\big] \wedge \big[\text{g}{:}T_2\big] \approx \begin{bmatrix}\text{f}{:}T_1 \\ \text{g}{:}T_2\end{bmatrix}$$

$$\big[\text{f}{:}T_1\big] \wedge \big[\text{f}{:}T_2\big] \approx \big[\text{f}{:}T_1 \wedge T_2\big]$$

We define a function μ which maps meets of record types to an equivalent record type, record types to equivalent types where meets in their values have been simplified by μ and any other types to themselves:

1. If for some T_1, T_2, $T = T_1 \wedge T_2$ then $\mu(T) = \mu'(\mu(T_1) \wedge \mu(T_2))$.
2. If T is a record type then $\mu(T)$ is T' such that for any ℓ,v, $\langle \ell, \mu(v) \rangle \in T'$ iff $\langle \ell, v \rangle \in T$.
3. Otherwise $\mu(T) = T$.

$\mu'(T_1 \wedge T_2)$ is defined by:

1. if T_1 and T_2 are record types, then $\mu'(T_1 \wedge T_2) = T_3$ such that
   (a) for any ℓ, v_1, v_2, if $\langle \ell, v_1 \rangle \in T_1$ and $\langle \ell, v_2 \rangle \in T_2$, then
      i. if v_1 and v_2 are $\langle \lambda u_1 : T'_1 \ldots \lambda u_i : T'_i(\phi), \langle \pi_1 \ldots \pi_i \rangle \rangle$ and $\langle \lambda u'_1 : T''_1 \ldots \lambda u'_k : T''_k(\psi), \langle \pi'_1 \ldots \pi'_k \rangle \rangle$ respectively, then $\langle \lambda u_1 : T'_1 \ldots \lambda u_i : T'_i, \lambda u'_1 : T''_1 \ldots \lambda u'_k : T''_k(\mu(\phi \wedge \psi)), \langle \pi_1 \ldots \pi_i, \pi'_1 \ldots \pi'_k \rangle \rangle \in T_3$
      ii. if v_1 is $\langle \lambda u_1 : T'_1 \ldots \lambda u_i : T'_i(\phi), \langle \pi_1 \ldots \pi_i \rangle \rangle$ and v_2 is a type (i.e. not of the form $\langle f, \Pi \rangle$ for some function f and sequence of paths Π), then $\langle \lambda u_1 : T'_1 \ldots \lambda u_i : T'_i(\mu(\phi \wedge v_2)), \langle \pi_1 \ldots \pi_i \rangle \rangle \in T_3$

iii. if v_2 is $\langle \lambda u'_1 : T''_1 \ldots \lambda u'_k : T''_k(\psi), \langle \pi'_1 \ldots \pi'_k \rangle \rangle$ and v_1 is a type, then $\langle \lambda u'_1 : T''_1 \ldots \lambda u'_k : T''_k(\mu(v_1 \wedge \psi)), \langle \pi'_1 \ldots \pi'_k \rangle \rangle \in T_3$

iv. otherwise $\langle \ell, \mu(v_1 \wedge v_2) \rangle \in T_3$

(b) for any ℓ, v_1, if $\langle \ell, v_1 \rangle \in T_1$ and there is no v_2 such that $\langle \ell, v_2 \rangle \in T_2$, then $\langle \ell, v_1 \rangle \in T_3$

(c) for any ℓ, v_2, if $\langle \ell, v_2 \rangle \in T_2$ and there is no v_1 such that $\langle \ell, v_1 \rangle \in T_1$, then $\langle \ell, v_2 \rangle \in T_3$

2. Otherwise $\mu'(T_1 \wedge T_2) = T_1 \wedge T_2$

$T_1 \mathbin{\dot{\wedge}} T_2$ is used to represent $\mu(T_1 \wedge T_2)$. We call $T_1 \mathbin{\dot{\wedge}} T_2$ the *merge* of T_1 and T_2.

The following two clauses could be added at the beginning of the definition of μ (after providing a characterization of the subtype relation, $\sqsubseteq$).

1. if for some T_1, T_2, $T = T_1 \wedge T_2$ and $T_1 \sqsubseteq T_2$ then $\mu(T) = T_1$
2. if for some T_1, T_2, $T = T_1 \wedge T_2$ and $T_2 \sqsubseteq T_1$ then $\mu(T) = T_2$

The current first clause would then hold in case neither of the conditions of these two clauses are met. The definition without these additional clauses only accounts for simplification of meets which have to do with merges of record types whereas the definition with the additional clauses would in addition have the effect, for example, that $\mu(T \wedge T_a) = T_a$ and $\mu(T_1 \wedge (T_1 \vee T_2)) = T_1$ (provided that we have an appropriate definition of $\sqsubseteq$) whereas the current definition without the additional clauses means that μ leaves these types unchanged.

We define also a notion of *asymmetric merge* of T_1 and T_2 which is defined by a function exactly like μ except that clause 2 of the definition of μ' is replaced by

2′. Otherwise $\mu'(T_1 \wedge T_2) = T_2$

We use $T_1 \boxed{\mathbin{\dot{\wedge}}} T_2$ to represent the asymmetric merge of T_1 and T_2.

These definitions do not in general avoid the formation of ill-formed record types since they allow record types to be replaced with non-record types within a record type thus potentially removing paths that might be included in dependent type fields elsewhere in the resulting type. However, if merging is restricted to either two record types or two non-record types this problem should not occur since all paths from both types will be preserved. In the case of asymmetric merges we can allow the replacement of non-record types by record types without risk. Note that our definition of dependent record types in A.11 allows for dependencies to fields that have conflicting types. Such record types will be well-formed though will not have any witnesses.

Merging functions which return types $\lambda r : T_1 \,.\, T_2(r) \mathbin{\dot{\wedge}} \lambda r : T_3 \,.\, T_4(r)$ denotes the function $\lambda r : T_1 \dot{\wedge} T_3 \,.\, T_2(r) \dot{\wedge} T_4(r)$.

A.13 Strings and regular types

A *string algebra over a set of objects* O is a pair $\langle S, ^\frown \rangle$ where:

1. S is the closure of $O \cup \{\varepsilon\}$ (ε is the empty string) under the binary operation '$^\frown$' ("concatenation")
2. for any s in S, $\varepsilon ^\frown s = s ^\frown \varepsilon = s$
3. for any s_1, s_2, s_3 in S, $(s_1 ^\frown s_2) ^\frown s_3 = s_1 ^\frown (s_2 ^\frown s_3)$. For this reason we normally write $s_1 ^\frown s_2 ^\frown s_3$ or more simply $s_1 s_2 s_3$.

The objects in S are called strings. Strings have length. ε has length 0. If s is a string in S with length n and a is an object in O then $s ^\frown a$ has length $n + 1$. We use $s[n]$ to represent the nth element of string s.

In TTR strings are records[9] with fields labelled by a distinguished ordered countably infinite set of labels (corresponding to the natural numbers): t_0,t_1,.... ε is the empty record, that is the empty set. This has length 0. (Recall that records are sets of ordered pairs.) A string with one element a (of some type) is the record $\left[\mathrm{t}_0{=}a\right]$. If s is a string whose highest label in the order is t_n ($n \geq 0$) and a is an object of some type then $s ^\frown a$ is $s \cup \{\langle t_{n+1}, a\rangle\}$. If s is a string whose highest label in the order is t_n then the length of s is $n + 1$. $s[n]$ is defined to be $s.\mathrm{t}_n$. Concatenation ('$^\frown$') can be extended to include concatenation of strings of arbitrary length. If s is a string and s_n is a string of length n, then $s ^\frown s_n$ is $s ^\frown s_n[0] ^\frown \ldots ^\frown s_n[n-1]$.

If s is a string of length n of records such that for each i, $0 \leq i < n$, $s[i].\pi$ is a defined path, $\underset{0 \leq i < n}{\mathrm{concat}}(s[i].\pi)$ denotes $s[0].\pi ^\frown \ldots ^\frown s[n{-}1].\pi$. We use $\mathrm{concat}_i(s[i].\pi)$ to represent $\underset{0 \leq i < \mathrm{length}(s)}{\mathrm{concat}}(s[i].\pi)$.

A system of complex types **TYPE**$_S$ = $\langle$**Type**, **BType**, $\langle$**PType**, **Pred**, **ArgIndices**, *Arity*$\rangle$, $\langle A, F\rangle\rangle$ with record types based on $\langle L, \mathbf{RType}\rangle$ *has string types* if

1. for each natural number i, $\mathrm{t}_i \in L$
2. *String* $\in$ **BType**

[9] This is new since Cooper (2012b).

3. $\emptyset \in$ *String*
4. if $T \in$ **Type** and $a :_{\mathbf{TYPE}_S} T$ then $\left[\text{t}_0\text{=a}\right]$: *String*
5. if $s :_{\mathbf{TYPE}_S}$ *String*, t_n is a label in s such that there is no $i > n$ where t_i is a label in s, $T \in$ **Type** and $a :_{\mathbf{TYPE}_S} T$ then $s \cup \{\langle \text{t}_{n+1}, a\rangle\} :_{\mathbf{TYPE}_S}$ *String*
6. Nothing is of type *String* except as required above.

We can define types whose elements are strings. Such types correspond to regular expressions and we will call them *regular types*. Here we will define just two kinds of such types: concatenation types and Kleene-+ types.

A system of complex types with string types **TYPE**$_S$ = $\langle$**Type**, **BType**, $\langle$**PType**, **Pred**, **ArgIndices**, *Arity*$\rangle$, $\langle A, F\rangle\rangle$ *has concatenation types* if

1. (a) for any $T_1, T_2 \in$ **Type**, $(T_1^\frown T_2) \in$ **Type**
   (b) for any $T_1, T_2, T_3 \in$ **Type**, $(T_1^\frown T_2)^\frown T_3 = T_1^\frown (T_2^\frown T_3)$[10]
2. $a :_{\mathbf{TYPE}_S} T_1^\frown T_2$ iff $a = x^\frown y$, $x :_{\mathbf{TYPE}_S} T_1$ and $y :_{\mathbf{TYPE}_S} T_2$

TYPE$_S$ *has Kleene-+ types* if

1. for any $T \in$ **Type**, $T^+ \in$ **Type**
2. $a :_{\mathbf{TYPE}_S} T^+$ iff $a = x_1^\frown \ldots^\frown x_n$, $n > 0$ and for i, $1 \leq i \leq n$, $x_i :_{\mathbf{TYPE}_S} T$

TYPE$_S$ *has Kleene-* types* if

1. for any $T \in$ **Type**, $T^* \in$ **Type**
2. $a :_{\mathbf{TYPE}_S} T^*$ iff $a = x_1^\frown \ldots^\frown x_n$, $n \geq 0$ and for i, $1 \leq i \leq n$, $x_i :_{\mathbf{TYPE}_S} T$

Note that this definition distinguishes and object a and the unit string consisting of a, that is, $\left[t_0\text{=}a\right]$. We will use $\widehat{a}$ to represent this.

Strings are used standardly in formal language theory where strings of symbols or strings of words are normally considered. Following important insights by Tim Fernando Fernando (2004, 2006, 2008, 2009) we shall be concerned rather with strings of events. We use informal notations like ‘ “sam” ’ and ‘ “ran” ’ to represent phonological types of speech events (utterances of *Sam* and *ran*). Thus ‘ “sam”$^\frown$“ran” ’ is the type of speech events which are concatenations of an utterance of *Sam* and an utterance of *ran*.

[10] This has been added to the definition in Cooper (2012b) to make associativity explicit.

Predicates which relate strings

We introduce a number of distinguished predicates which are used to relate strings. The following predicates all have arity [*String*,*String*]: init, final, final_align

init "s_1 is an initial substring of s_2"

If s_1 is a string of length n and s_2 is a string of any length, then s : init(s_1,s_2) iff the length of s_2 is greater than or equal to n and for each i, $0 \leq i < n$, $s_1[i] = s_2[i]$ and $s = s_2$.

final "s_1 is an final substring of s_2"

If s_1 is a string of length n and s_2 is a string of length m, then s : final(s_1,s_2) iff m is greater than or equal to n and for each i, $0 \leq i < n$, $s_1[i] = s_2[(m-n)+i]$ and $s = s_2$.

final_align "s_1 is aligned with a final substring of s_2"

If s_1:*Rec*$^+$ is a string of length n and s_2:*Rec*$^+$ is a string of length m, then s : final_align(s_1,s_2) iff

1. m is greater than or equal to n
2. s is a string of length m
3. for each i, $0 \leq i < n$,
   (a) $s[(m-n)+i] : \begin{bmatrix} \mathrm{e}_1\text{:}\mathit{Rec} \\ \mathrm{e}_2\text{:}\mathit{Rec} \end{bmatrix}$
   (b) $s[(m-n)+i].\mathrm{e}_1 = s_1[i]$
   (c) $s[(m-n)+i].\mathrm{e}_2 = s_2[(m-n)+i]$
4. otherwise for each i, $0 \leq i < m$, $s[i] = s_2[i]$

Appendix B

Grammar rules

B.1 Universal resources

B.1.1 Signs

Sign (Chapter 2)

$$\begin{bmatrix} \text{s-event} & : & \mathit{SEvent} \\ \text{cnt} & : & \mathit{Cnt} \end{bmatrix}$$

Sign (Chapter 3)

a recursive type

$$\sigma : \mathit{Sign} \text{ iff } \sigma : \begin{bmatrix} \text{s-event} & : & \mathit{SEvent} \\ \text{syn} & : & \mathit{Syn} \\ \text{cnt} & : & \mathit{Cnt} \end{bmatrix}$$

SEvent (Chapter 2)

$$\left[\begin{array}{lcl} \text{e-loc} & : & \textit{Loc} \\ \text{sp} & : & \textit{Ind} \\ \text{au} & : & \textit{Ind} \\ \text{e} & : & \textit{Phon} \\ \text{c}_{\text{loc}} & : & \text{loc(e,e-loc)} \\ \text{c}_{\text{sp}} & : & \text{speaker(e,sp)} \\ \text{c}_{\text{au}} & : & \text{audience(e,au)} \end{array}\right]$$

Phon (Chapter 2)

$\textit{Word}^{+}$

Cnt (Chapter 2)

RecType

Cnt (Chapter 3)

$\textit{RecType} \vee \textit{Ppty} \vee \textit{Quant} \vee (\textit{Ppty} \rightarrow \textit{Quant})$

Ppty (Chapter 3)

$(\left[\text{x:}\textit{Ind}\right] \rightarrow \textit{RecType})$

Quant (Chapter 3)

$(\textit{Ppty} \rightarrow \textit{RecType})$

Syn (Chapter 3)

$$\left[\begin{array}{lcl} \text{cat} & : & \textit{Cat} \\ \text{daughters} & : & \textit{Sign}^{*} \end{array}\right]$$

Cat (Chapter 3)

s, np, det, n, v, vp : *Cat*

Category sign types:

S (Chapter 3)
$\textit{Sign} \wedge\!\!\!\cdot\ \left[\text{syn:}\left[\text{cat=s:}\textit{Cat}\right]\right]$
NP (Chapter 3)

Sign $\wedge\!\!\!\cdot$ [syn:[cat=np:*Cat*]]

Det (Chapter 3)

Sign $\wedge\!\!\!\cdot$ [syn:[cat=det:*Cat*]]

N (Chapter 3)

Sign $\wedge\!\!\!\cdot$ [syn:[cat=n:*Cat*]]

V (Chapter 3)

Sign $\wedge\!\!\!\cdot$ [syn:[cat=v:*Cat*]]

VP (Chapter 3)

Sign $\wedge\!\!\!\cdot$ [syn:[cat=vp:*Cat*]]

NoDaughters (Chapter 3)

[syn:[daughters=ε:*Sign**]]

B.1.2 Sign type construction operations

Lexicon

sign (Chapter 2)

If σ is a type of speech event and κ is a type (of situation) then

$$\text{sign}(\sigma,\kappa)= \begin{bmatrix} \text{s-event:}[\text{e:}\sigma] \\ \text{cnt=}\begin{bmatrix} \text{e:}\kappa \\ \text{c}_{\text{tns}}\text{:final\_align}(\Uparrow\text{s-event.e,e}) \end{bmatrix}\text{:}\mathit{RecType} \end{bmatrix}$$

sign$_{uc}$ (Chapter 2)

If σ is a type of speech event then

$$\text{sign}_{uc}(\sigma)= \begin{bmatrix} \text{s-event:}[\text{e:}\sigma] \\ \text{cnt:}\mathit{RecType} \end{bmatrix}$$

Lex (Chapter 3)

λT_1:*Type*
λT_2:*Type* .
$T_1 \wedge\!\!\!\cdot$ [s-event:[e:T_2]] $\wedge\!\!\!\cdot$ *NoDaughters*

Licensing condition associated with lexical resources (Chapter 3)

If Lex(T, C) is a resource available to agent A, then for any u, $u :_A T$ licenses $:_A$ Lex(T, C) $\wedge_{\cdot} [\text{s-event:} [\text{e=}u\text{:}T]]$

Universal resources for lexical content construction

SemCommonNoun(p), where p is a predicate with arity $\langle \mathit{Ind} \rangle$ (Chapter 3)

$\lambda r\text{:}[\text{x:}\mathit{Ind}] \,.\, [\ \text{e} \ : \ p(r.\text{x})\ ]$

SemPropName(a), where a:*Ind* (Chapter 3)

$\lambda P\text{:}\mathit{Ppty} \,.\, P([\text{x=}a])$

SemIndefArt (Chapter 3)

$$\lambda Q\text{:}\mathit{Ppty} \,.\, \lambda P\text{:}\mathit{Ppty} \,.\, \begin{bmatrix} \text{restr=}Q & : & \mathit{Ppty} \\ \text{scope=}P & : & \mathit{Ppty} \\ \text{e} & : & \text{exist(restr, scope)} \end{bmatrix}$$

SemBe (Chapter 3)

$\lambda \mathcal{Q}\text{:}\mathit{Quant} \,.\, \lambda r_1\text{:}[\text{x:}\mathit{Ind}] \,.\, \mathcal{Q}(\lambda r_2\text{:}[\text{x:}\mathit{Ind}] \,.\, [\ \text{e} \ : \ r_2.\text{x} = r_1.\text{x}\ ])$

Universal resources for associating lexical content with phonological types

$\text{Lex}_{\text{PropName}}(T_{\text{Phon}}, a)$, where T_{Phon} is a phonological type and a:*Ind* (Chapter 3)
Lex(T_{Phon}, *NP*) $\wedge_{\cdot} [\text{cnt=SemPropName}(a)\text{:}\mathit{Quant}]$

$\text{Lex}_{\text{IndefArt}}(T_{\text{Phon}})$, where T_{Phon} is a phonological type (Chapter 3)
Lex(T_{Phon}, *Det*) $\wedge_{\cdot} [\text{cnt=SemIndefArt:}(\mathit{Ppty} \rightarrow \mathit{Quant})]$

$\text{Lex}_{\text{be}}(T_{\text{Phon}})$, where T_{Phon} is a phonological type (Chapter 3)
Lex(T_{Phon}, *V*) $\wedge_{\cdot} [\text{cnt=SemBe:}(\mathit{Quant} \rightarrow \mathit{Ppty})]$

Operations which construct sign combination functions

Licensing condition associated with sign combination functions (Chapter 3)

If $f : (T_1 \rightarrow Type)$ is a sign combination function available to agent A, then for any u, $u :_A T_1$ licenses $:_A f(u)$

RuleDaughters (Chapter 3)

RuleDaughters maps two types to a sign combination function

λT_1 : ***Type***
  λT_2 : ***Type*** .
    $\lambda u : T_1 \,.\, T_2 \wedge \left[\text{syn:}\left[\text{daughters=}u{:}T_1\right]\right]$

ConcatPhon (Chapter 3)

$\lambda u{:}\left[\text{s-event:}\left[\text{e:}\textit{Phon}\right]\right]^+$.
  [s-event : [e=concat$_i$($u[i]$.s-event.e) : *Phon*]]

Phrase structure rule notation (Chapter 3)

If $C, C_1, \ldots, C_n$ are category sign types then,

$C \longrightarrow C_1 \ldots C_n$ represents RuleDaughters(C, $C_1 ^\frown \ldots ^\frown C_n$) $\wedge$ ConcatPhon

CntForwardApp (Chapter 3)

λT_1:***Type*** λT_2:***Type*** .
  $\lambda u{:}\left[\text{cnt:}(T_2 \rightarrow T_1)\right] ^\frown \left[\text{cnt:}T_2\right]$.
    $\left[\text{cnt=}u\text{[0].cnt}(u\text{[1].cnt}){:}T_1\right]$

B.2 English resources

B.2.1 Lexicon

(Chapter 2)

sign(“Dudamel is a conductor”, conductor(dudamel)),
sign(“Beethoven is a composer”, composer(beethoven)),
sign(“Uchida is a pianist”, pianist(uchida)),
sign_{uc}(“ok”),
sign_{uc}(“aha”)

(Chapter 3)

Lex(“Dudamel”, *NP*)
Lex(“Beethoven”, *NP*)
Lex(“a”, *Det*)
Lex(“composer”, *N*)
Lex(“conductor”, *N*)
Lex(“is”, *V*)
Lex(“ok”, *S*)
Lex(“aha”,*S*)

$\text{Lex}_{\text{PropName}}$(“Dudamel”, d), where d:*Ind*
$\text{Lex}_{\text{PropName}}$(“Beethoven”, b), where b:*Ind*
$\text{Lex}_{\text{CommonNoun}}$(“composer”, composer), where ‘composer’ is a predicate with arity $\langle[\text{x:}\mathit{Ind}]\rangle$
$\text{Lex}_{\text{CommonNoun}}$(“conductor”, conductor), where ‘conductor’ is a predicate with arity $\langle[\text{x:}\mathit{Ind}]\rangle$
$\text{Lex}_{\text{IndefArt}}$(“a”)
Lex_{be}(“is”)

B.2.2 Phrase structure

(Chapter 3)

$S \longrightarrow NP\ VP$
$NP \longrightarrow Det\ N$
$VP \longrightarrow V\ NP$

B.2.3 Non-compositional Constructions

CnstrIsA (Chapter 3)

$$\lambda u{:}V \mathbin{\dot\wedge} [\text{s-event:}[\text{e:“is”}]] \frown NP \mathbin{\dot\wedge} \left[\text{syn:}\left[\begin{array}{l}\text{daughters:}\mathit{Det} \mathbin{\dot\wedge} [\text{s-event:}[\text{e:“a”}]] \\ \qquad\qquad\qquad \frown N \mathbin{\dot\wedge} [\text{cnt:}\mathit{Ppty}]\end{array}\right]\right].$$
$$VP \mathbin{\dot\wedge} [\text{cnt=}u[2].\text{syn.daughters}[2].\text{cnt:}\mathit{Ppty}]$$

B.2.4 Interpreted phrase structure

(Chapter 3)

$S \longrightarrow NP\ VP \underset{\vdots}{\wedge}$ CntForwardApp(*Ppty*, *RecType*)

$NP \longrightarrow Det\ N \underset{\vdots}{\wedge}$ CntForwardApp(*Ppty*, *Quant*)

$VP \longrightarrow V\ NP \underset{\vdots}{\wedge}$ CnstrIsA

A more readable abbreviatory notation for these rules is:

$S \longrightarrow NP\ VP \mid NP'(VP')$

$NP \longrightarrow Det\ N \mid Det'(N')$

$VP \longrightarrow [_V$ “is”] $[_{NP}\ [_{Det}$ “a”] $N] \mid N'$

Note that this last rule does not correspond to a context-free phrase-structure rule.

Appendix C

Dialogue rules

C.1 Universal resources

C.1.1 Types of Information States

InfoState (Chapter 2)

$$\begin{bmatrix} \text{private:} \begin{bmatrix} \text{agenda:}[\mathit{MoveType}(\mathit{SELF})] \end{bmatrix} \\ \text{shared:} \begin{bmatrix} \text{latest-utterance:} \begin{bmatrix} \text{move:}\mathit{Move}(\mathit{SELF}) \\ \text{chart:}\mathit{Chart} \\ \text{e:m-interp(chart,move)} \end{bmatrix} \vee \mathit{Nil} \\ \text{commitments:}\mathit{RecType} \end{bmatrix} \end{bmatrix}$$

InitInfoState (Chapter 2)

The type of initial or empty information states

$$\begin{bmatrix} \text{private:} \begin{bmatrix} \text{agenda=[]:}[\mathit{RecType}] \end{bmatrix} \\ \text{shared:} \begin{bmatrix} \text{latest-utterance:}\mathit{Nil} \\ \text{commitments=[]:}\mathit{RecType} \end{bmatrix} \end{bmatrix}$$

C.1.2 Action functions

Licensing conditions on type acts If $f : (T \rightarrow \textit{Type})$ is an action function then for any object a and agent A, $a :_A T$ licenses $:_A f(a)$!

de se: If $f : (T \rightarrow (\textit{Ind} \rightarrow \textit{Type}))$ is an action function then for any object a and agent A, $a :_A T$ licenses $:_A f(a)(A)$!

ExecTopAgenda (Chapter 2)

$$\lambda r{:}\left[\begin{array}{lcl}\text{private} & : & \left[\begin{array}{lcl}\text{agenda} & : & {}_{ne}[\textit{RecType}]\end{array}\right]\end{array}\right] \;.\; \left[\begin{array}{lcl}\text{move} & : & \text{fst}(r.\text{private.agenda}) \\ \text{chart} & : & \textit{Chart} \\ \text{e} & : & \text{m-interp(chart,move)}\end{array}\right]$$

C.1.3 Perception functions (type shifts)

Licensing conditions on type acts If $f : (T \rightarrow \textit{Type})$ is a perception function then for any object o and agent A, $o :_A T$ licenses $o :_A f(o)$

de se: If $f : (T \rightarrow (\textit{Ind} \rightarrow \textit{Type}))$ is a perception function then for any object o and agent A, $o :_A T$ licenses $o :_A f(o)(A)$

PerceiveSpeechAct(T), $T \sqsubseteq \textit{Phon}$ (Chapter 2)

$$\lambda e{:}\left[\begin{array}{l}\text{e:}T \\ \text{au=}\textit{SELF}\text{:}\textit{Ind}\end{array}\right].\; \left[\begin{array}{lcl}\text{move} & : & \left[\begin{array}{lcl}\text{e} & : & \textit{SpeechAct} \wedge\!\!\!\cdot \left[\text{au=}\textit{SELF}\text{:}\textit{Ind}\right] \\ \text{cnt} & : & \textit{Cnt} \\ \text{c}_{\text{cnt}} & : & \text{content(e,cnt)}\end{array}\right] \\ \text{chart} & : & \mathfrak{C}_T \\ \text{e} & : & \text{m-interp(chart,move)}\end{array}\right]$$

where $\mathfrak{C}_T$ is the type of charts assigned to utterances of type T (as a result of parsing). In Chapter 2 $\mathfrak{C}_T$ is equated with Σ_T, the type of signs associated with utterances of type T.

C.1.4 Update functions

Licensing conditions on type acts If $f : (T_1 \rightarrow (T_2 \rightarrow \mathit{Type}))$ is an update function, A is an agent, s_i is A's current information state, $s_i :_A T_i$, $T_i \sqsubseteq T_1$ (and $s_i : T_1$), then an event $e :_A T_2$ (and $e : T_2$) licenses $s_{i+1} :_A T_i \boxed{\wedge\!\!\!\cdot} f(s_i)(e)$.

IntegrateOwnAssertion (Chapter 2)

$$\lambda r{:}\left[\text{ private : }\left[\text{ agenda : }{}_{ne}[\mathit{MoveType}(\mathit{SELF})]\ \right]\ \right]$$
$$\lambda u{:}\left[\begin{array}{lll}\text{move} & : & \text{fst}(r.\text{private.agenda}) \wedge\!\!\!\cdot \left[\text{e:}\left[\begin{array}{l}\text{sp=}\mathit{SELF}{:}\mathit{Ind}\\ \text{au:}\mathit{Ind}\end{array}\right]\right] \wedge\!\!\!\cdot \left[\text{e:}\mathit{Assertion}\right]\\ \text{chart} & : & \mathit{Chart}\\ \text{e} & : & \text{m-interp(chart,move)}\end{array}\right].$$
$$\left[\text{private:}\left[\text{agenda=}\left[\begin{array}{l}\text{e:}\mathit{Acknowledgement} \wedge\!\!\!\cdot \left[\begin{array}{l}\text{sp=}u.\text{move.e.au:}\mathit{Ind}\\ \text{au=}\mathit{SELF}{:}\mathit{Ind}\end{array}\right]\\ \text{cnt=}u.\text{move.cnt:}\mathit{RecType}\\ \text{c}_{\text{cnt}}\text{:content(e,cnt)}\end{array}\right] \mid \text{rst}(r.\text{private.agenda})\right] :[\mathit{MoveType}(\mathit{SELF})]\right]\right.$$
$$\left.\left[\text{shared:}\left[\text{latest-utterance:}\left[\begin{array}{l}\text{move=}u.\text{move:}\mathit{Move}(\mathit{SELF})\\ \text{chart=}u.\text{chart:}\mathit{Chart}\\ \text{e=}u.\text{e:m-interp(chart,move)}\end{array}\right]\right]\right]\right]$$

IntegrateOtherAssertion (Chapter 2)

$$\lambda r{:}\left[\text{ private : }\left[\text{ agenda : }[\mathit{RecType}]\ \right]\ \right]$$
$$\lambda u{:}\left[\begin{array}{l}\text{move:}\left[\begin{array}{l}\text{e:}\mathit{Assertion} \wedge\!\!\!\cdot \left[\begin{array}{l}\text{sp:}\mathit{Ind}\\ \text{au=}\mathit{SELF}{:}\mathit{Ind}\end{array}\right]\\ \text{cnt:}\mathit{RecType}\\ \text{c}_{\text{cnt}}\text{:content(e,cnt)}\end{array}\right]\\ \text{chart:}\mathit{Chart}\\ \text{e:m-interp(chart,move)}\end{array}\right].$$
$$\left[\text{private:}\left[\text{agenda=}\left[\begin{array}{l}\text{e:}\mathit{Acknowledgement} \wedge\!\!\!\cdot \left[\begin{array}{l}\text{sp=}\mathit{SELF}{:}\mathit{Ind}\\ \text{au=}u.\text{move.e.sp:}\mathit{Ind}\end{array}\right]\\ \text{cnt=}u.\text{move.cnt:}\mathit{RecType}\\ \text{c}_{\text{cnt}}\text{:content(e,cnt)}\end{array}\right] \mid r.\text{private.agenda}\right] :[\mathit{RecType}]\right]\right.$$
$$\left.\left[\text{shared:}\left[\text{latest-utterance:}\left[\begin{array}{l}\text{move=}u.\text{move:}\mathit{Move}\\ \text{chart=}u.\text{chart:}\mathit{Chart}\\ \text{e=}u.\text{e:m-interp(chart,move)}\end{array}\right]\right]\right]\right]$$

IntegrateOwnAcknowledgement (Chapter 2)

$$\lambda r{:}\left[\begin{array}{lcl}\text{private} & : & \left[\text{agenda} : {}_{ne}[\textit{RecType}]\right]\\ \text{shared} & : & \left[\begin{array}{lcl}\text{latest-utterance} & : & \left[\text{move} : \left[\text{content} : \textit{RecType}\right]\right]\\ \text{commitments} & : & \textit{RecType}\end{array}\right]\end{array}\right]$$
$$\lambda u{:}\left[\begin{array}{lcl}\text{move} & : & \text{fst}(r.\text{private.agenda}) \dot{\wedge} \left[\text{e:}\textit{Acknowledgement}\right] \dot{\wedge} \left[\text{e:}\left[\text{sp=}\textit{SELF}\text{:}\textit{Ind}\right]\right]\\ \text{chart} & : & \textit{Chart}\\ \text{e} & : & \text{m-interp(chart,move)}\end{array}\right].$$
$$\left[\begin{array}{l}\text{private:}\left[\text{agenda=rst}(r.\text{private.agenda}){:}[\textit{RecType}]\right]\\ \text{shared:}\left[\begin{array}{l}\text{latest-utterance:}\left[\begin{array}{l}\text{move=}u.\text{move:}\textit{Move}\\ \text{chart=}u.\text{chart:}\textit{Chart}\\ \text{e=}u.\text{e:m-interp(chart,move)}\end{array}\right]\\ \text{commitments=}\left[\text{prev:}r.\text{commitments}\right] \dot{\wedge} u.\text{move.cnt:}\textit{RecType}\end{array}\right]\end{array}\right]$$

IntegrateOtherAcknowledgement (Chapter 2)

$$\lambda r{:}\left[\begin{array}{lcl}\text{private} & : & \left[\text{agenda} : {}_{ne}[\textit{RecType}]\right]\\ \text{shared} & : & \left[\begin{array}{lcl}\text{latest-utterance} & : & \left[\text{move} : \left[\text{content} : \textit{RecType}\right]\right]\\ \text{commitments} & : & \textit{RecType}\end{array}\right]\end{array}\right]$$
$$\lambda u{:}\left[\begin{array}{lcl}\text{move} & : & \text{fst}(r.\text{private.agenda}) \dot{\wedge} \left[\text{e:}\textit{Acknowledgement}\right] \dot{\wedge} \left[\text{e:}\left[\text{au=}\textit{SELF}\text{:}\textit{Ind}\right]\right]\\ \text{chart} & : & \textit{Chart}\\ \text{e} & : & \text{m-interp(chart,move)}\end{array}\right].$$
$$\left[\begin{array}{l}\text{private:}\left[\text{agenda=rst}(r.\text{private.agenda}){:}[\textit{RecType}]\right]\\ \text{shared:}\left[\begin{array}{l}\text{latest-utterance:}\left[\begin{array}{l}\text{move=}u.\text{move:}\textit{Move}\\ \text{chart=}u.\text{chart:}\textit{Chart}\\ \text{e=}u.\text{e:m-interp(chart,move)}\end{array}\right]\\ \text{commitments=}\left[\text{prev:}r.\text{commitments}\right] \dot{\wedge} u.\text{move.cnt:}\textit{RecType}\end{array}\right]\end{array}\right]$$

C.2 English resources

Bibliography

Artstein, Ron, Mark Core, David DeVault, Kallirroi Georgila, Elsi Kaiser and Amanda Stent, eds. (2011) SemDial 2011 (Los Angelogue): Proceedings of the 15th Workshop on the Semantics and Pragmatics of Dialogue.

Austin, J. (1962) *How to Do Things with Words*, Oxford University Press, ed. by J. O. Urmson.

Austin, J. L. (1961) Truth, in J. O. Urmson and G. J. Warnock (eds.), *J. L. Austin: Philosophical Papers*, Oxford University Press, Oxford.

Barwise, Jon (1989) *The Situation in Logic*, CSLI Publications, Stanford.

Barwise, Jon and Robin Cooper (1981) Generalized quantifiers and natural language, *Linguistics and Philosophy*, Vol. 4, No. 2, pp. 159–219.

Barwise, Jon and John Perry (1983) *Situations and Attitudes*, Bradford Books, MIT Press, Cambridge, Mass.

Blackburn, Patrick, Maarten de Rijke and Ydes Venema (2001) *Modal logic* (*Cambridge Tracts in Theoretical Computer Science* 53), Cambridge University Press.

Boas, Hans C. and Ivan A. Sag, eds. (2012) *Sign-Based Construction Grammar*, CSLI Publications.

Botvinick, Matthew M. (2008) Hierarchical models of behavior and prefrontal function, *Trends in Cognitive Sciences*, Vol. 12, No. 5, pp. 201 – 208.

Botvinick, Matthew M., Yael Niv and Andrew C. Barto (2009) Hierarchically organized behavior and its neural foundations: A reinforcement learning perspective, *Cognition*, Vol. 113, No. 3, pp. 262 – 280. Reinforcement learning and higher cognition.

Breitholtz, Ellen (2010) Clarification Requests as Enthymeme Elicitors, in *Aspects of Semantics and Pragmatics of Dialogue. SemDial 2010, 14th Workshop on the Semantics and Pragmatics of Dialogue ,*.

Breitholtz, Ellen and Robin Cooper (2011) Enthymemes as Rhetorical Resources, in Artstein *et al.* (2011).

Breitholtz, Ellen and Jessica Villing (2008) Can Aristotelian Enthymemes Decrease the Cognitive Load of a Dialogue System User?, in *Proceedings of LonDial 2008, the 12th SEMDIAL workshop*.

Bullock, Barbara E. and Almeida Jacqueline Toribio, eds. (2009) *The Cambridge Handbook of Linguistic Code-Switching*, Cambridge University Press.

Carnap, Rudolf (1956) *Meaning and Necessity: A Study in Semantics and Modal Logic*, second edition, University of Chicago Press.

Chierchia, Gennaro and Raymond Turner (1988) Semantics and property theory, *Linguistics and Philosophy*, Vol. 11, No. 3, pp. 261–302.

Cooper, Robin (1982) Binding in wholewheat* syntax (*unenriched with inaudibilia), in P. Jacobson and G. K. Pullum (eds.), *The Nature of Syntactic Representation* (*Synthese Language Library* 15), Reidel Publishing Company.

Cooper, Robin (2005) Records and Record Types in Semantic Theory, *Journal of Logic and Computation*, Vol. 15, No. 2, pp. 99–112.

Cooper, Robin (2011) Copredication, Quantfication and Frames, in S. Pogodalla and J.-P. Prost (eds.), *Logical Aspects of Computational Linguistics: 6th International Conference, LACL 2011*, pp. 64–79, Springer.

Cooper, Robin (2012a) Intensional quantifiers, in T. Graf, D. Paperno, A. Szabolcsi and J. Tellings (eds.), *Theories of Everything: In Honor of Ed Keenan*, UCLA Working Papers in Linguistics *17*, pp. 69–71, Department of Linguistics, UCLA.

Cooper, Robin (2012b) Type Theory and Semantics in Flux, in R. Kempson, N. Asher and T. Fernando (eds.), *Handbook of the Philosophy of Science*, Vol. 14: Philosophy of Linguistics, pp. 271–323, Elsevier BV. General editors: Dov M. Gabbay, Paul Thagard and John Woods.

Cooper, Robin (2013) Clarification and Generalized Quantifiers, *Dialogue and Discourse*, Vol. 4, No. 1, pp. 1–25.

Cooper, Robin (fthc) Type Theory and Semantics in Flux, in R. Kempson, N. Asher and T. Fernando (eds.), *Handbook of the Philosophy of Science*, Vol. 14: Philosophy of Linguistics, Elsevier BV. General editors: Dov M. Gabbay, Paul Thagard and John Woods.

Cooper, Robin and Jonathan Ginzburg (2011a) Negation in Dialogue, in Artstein *et al.* (2011), pp. 130–139.

Cooper, Robin and Jonathan Ginzburg (2011b) Negative inquisitiveness and alternatives-based negation, in *Proceedings of the Amsterdam Colloquium, 2011*.

Cresswell, M.J. (1985) *Structured Meanings: The Semantics of Propositional Attitudes*, MIT Press.

Davidson, Donald (1967) The Logical Form of Action Sentences, in N. Rescher (ed.), *The Logic of Decision and Action*, University of Pittsburgh Press. Reprinted in Davidson (1980).

Davidson, Donald (1980) *Essays on Actions and Events*, Oxford University Press, New edition 2001.

Dowty, David (1989) On the semantic content of the notion of 'Thematic Role', in G. Chierchia, B. H. Partee and R. Turner (eds.), *Properties, Types and Meanings*, Vol. II: Semantic Issues, pp. 69–130, Kluwer, Dordrecht.

Dowty, David, Robert Wall and Stanley Peters (1981) *Introduction to Montague Semantics*, Reidel (Springer).

Fernández, Raquel (2006) *Non-sentential utterances in dialogue: Classification, Resolution and Use*, PhD dissertation, King's College, London.

Fernando, Tim (2004) A finite-state approach to events in natural language semantics, *Journal of Logic and Computation*, Vol. 14, No. 1, pp. 79–92.

Fernando, Tim (2006) Situations as Strings, *Electronic Notes in Theoretical Computer Science*, Vol. 165, pp. 23–36.

Fernando, Tim (2008) Finite-state descriptions for temporal semantics, in H. Bunt and R. Muskens (eds.), *Computing Meaning, Volume 3* (*Studies in Linguistics and Philosophy* 83), pp. 347–368, Springer.

Fernando, Tim (2009) Situations in LTL as strings, *Information and Computation*, Vol. 207, No. 10, pp. 980–999.

Fernando, Tim (2011) Constructing Situations and Time, *Journal of Philosophical Logic*, Vol. 40, pp. 371–396.

Fillmore, Charles J. (1982) Frame semantics, in *Linguistics in the Morning Calm*, pp. 111–137, Hanshin Publishing Co., Seoul.

Fillmore, Charles J. (1985) Frames and the semantics of understanding, *Quaderni di Semantica*, Vol. 6, No. 2, pp. 222–254.

Fox, Chris and Shalom Lappin (2005) *Foundations of Intensional Semantics*, Blackwell Publishing.

Gibson, James J. (1986) *The Ecological Approach to Visual Perception*, Lawrence Erlbaum Associates.

Gil, David (2000) Syntactic categories, cross-linguistic variation and universal grammar, in P. M. Vogel and B. Comrie (eds.), *Approaches to the typology of word classes* (*Empirical approaches to language typology* 23), Mouton de Gruyter, Berlin.

Ginzburg, Jonathan (1994) An update semantics for dialogue, in H. Bunt (ed.), *Proceedings of the 1st International Workshop on Computational Semantics*, Tilburg University.

Ginzburg, Jonathan (2010) Relevance for Dialogue, in Łupkowski and Purver (2010), pp. 121–129, Polish Society for Cognitive Science.

Ginzburg, Jonathan (2012) *The Interactive Stance: Meaning for Conversation*, Oxford University Press, Oxford.

Ginzburg, Jonathan (fthc) *The Interactive Stance: Meaning for Conversation*, Oxford University Press, Oxford.

Ginzburg, Jonathan and Raquel Fernández (2010) Computational Models of Dialogue, in A. Clark, C. Fox and S. Lappin (eds.), *The Handbook of Computational Linguistics and Natural Language Processing*, Wiley-Blackwell.

Ginzburg, Jonathan and Ivan A. Sag (2000) *Interrogative Investigations: The Form, Meaning, and Use of English Interrogatives*, CSLI Lecture Notes *123*, CSLI Publications, Stanford, California.

Groenendijk, Jeroen and Floris Roelofsen (2012) Course Notes on Inquisitive Semantics, NASSLLI 2012. Available at `https://sites.google.com/site/inquisitivesemantics/documents/NASSLLI-2012-inquisitive-semantics-lecture-notes.pdf`.

Heim, Irene and Angelika Kratzer (1998) *Semantics in Generative Grammar*, Blackwell Publishing.

Hughes, G.E. and M.J. Cresswell (1968) *Introduction to modal logic*, Methuen and Co., Ltd.

Jackendoff, Ray (2002) *Foundations of Language: Brain, Meaning, Grammar, Evolution*, Oxford University Press.

Jurafsky, Daniel and James H. Martin (2009) *Speech and Language Processing*, second edition, Pearson Education.

Kamp, Hans and Uwe Reyle (1993) *From Discourse to Logic*, Kluwer, Dordrecht.

Kant, Immanuel (1781) *Critik der reinen Vernunft (Critique of Pure Reason)*, Johann Friedrich Hartknoch, Riga, second edition 1787.

Keenan, E. L. and J. Stavi (1986) Natural Language Determiners, *Linguistics and Philosophy*, Vol. 9, pp. 253–326.

Larsson, Staffan (2002) *Issue-based Dialogue Management*, PhD dissertation, University of Gothenburg.

Larsson, Staffan (2011) The TTR perceptron: Dynamic perceptual meanings and semantic coordination., in Artstein *et al.* (2011).

Larsson, Staffan and David R. Traum (2001) Information state and dialogue management in the TRINDI dialogue move engine toolkit, *Natural Language Engineering*, Vol. 6, No. 3&4, pp. 323–340.

Lewis, David (1972) General Semantics, in D. Davidson and G. Harman (eds.), *Semantics of Natural Language*, pp. 169–218, Reidel Publishing Company.

Lewis, David (1979) Attitudes de dicto and de se, *Philophical Review*, Vol. 88, pp. 513–543. Reprinted in Lewis (1983).

Lewis, David (1983) *Philosophical Papers, Volume 1*, Oxford University Press.

Łupkowski, Paweł and Matthew Purver, eds. (2010) Aspects of Semantics and Pragmatics of Dialogue. SemDial 2010, 14th Workshop on the Semantics and Pragmatics of Dialogue. Poznań: Polish Society for Cognitive Science.

Martin-Löf, Per (1984) *Intuitionistic Type Theory*, Bibliopolis, Naples.

McCarthy, J. and P. J. Hayes (1969) Some philosophical problems from the standpoint of artificial intelligence, *Machine Intelligence*, Vol. 4, pp. 463–502.

Montague, Richard (1973) The Proper Treatment of Quantification in Ordinary English, in J. Hintikka, J. Moravcsik and P. Suppes (eds.), *Approaches to Natural Language: Proceedings of the 1970 Stanford Workshop on Grammar and Semantics*, pp. 247–270, D. Reidel Publishing Company, Dordrecht.

Montague, Richard (1974) *Formal Philosophy: Selected Papers of Richard Montague*, Yale University Press, New Haven, ed. and with an introduction by Richmond H. Thomason.

Ninan, Dilip (2010) *De Se* Attitudes: Ascription and Communication, *Philosophy Compass*, Vol. 5, No. 7, pp. 551–567.

Nordström, Bengt, Kent Petersson and Jan M. Smith (1990) *Programming in Martin-Löf's Type Theory* (*International Series of Monographs on Computer Science* 7), Clarendon Press, Oxford.

Partee, B.H., A.G.B. ter Meulen and R.E. Wall (1990) *Mathematical Methods in Linguistics*, Springer.

Partee, Barbara H. (1986) Noun Phrase Interpretation and Type-Shifting Principles, in J. Groenendijk, D. de Jongh and M. Stokhof (eds.), *Studies in Discourse Representation Theory and the Theory of Generalized Quantifiers*, Foris Publications.

Partee, Barbara H. and Vladimir Borschev (2012) Sortal, Relational, and Functional Interpretations of Nouns and Russian Container Constructions, *Journal of Semantics*, Vol. 29, No. 4, pp. 445–486.

Perry, John (1977) The Problem of the Essential Indexical, *Noûs*, Vol. 13, No. 1, pp. 3–21. Reprinted in Perry (1993).

Perry, John (1993) *The Problem of the Essential Indexical and Other Essays*, Oxford University Press.

Peters, Stanley and Dag Westerståhl (2006) *Quantifiers in Language and Logics*, Oxford University Press.

Prior, Arthur N. (1957) *Time and modality*, Oxford University Press.

Prior, Arthur N. (1967) *Past, present and future*, Oxford University Press.

Prior, Arthur N. (2003) *Papers on Time and Tense*, Oxford University Press, new edition, edited by Per Hasle, Peter Øhrstrøm, Torbren Braüner and Jack Copeland.

Purver, Matthew, Eleni Gregoromichelaki, Wilfried Meyer-Viol and Ronnie Cann (2010) Splitting the *I*s and Crossing the *You*s: Context, Speech Acts and Grammar, in Łupkowski and Purver (2010), pp. 43–50, Polish Society for Cognitive Science.

Pustejovsky, James (1995) *The Generative Lexicon*, MIT Press, Cambridge, Mass.

Pustejovsky, James (2006) Type theory and lexical decomposition, *Journal of Cognitive Science*, Vol. 6, pp. 39–76.

Ranta, Aarne (1994) *Type-Theoretical Grammar*, Clarendon Press, Oxford.

Reichenbach, Hans (1947) *Elements of Symbolic Logic*, University of California Press.

Ribas-Fernandes, José J.F., Alec Solway, Carlos Diuk, Joseph T. McGuire, Andrew G. Barto, Yael Niv and Matthew M. Botvinick (2011) A Neural Signature of Hierarchical Reinforcement Learning, *Neuron*, Vol. 71, No. 2, pp. 370 – 379.

Ruppenhofer, Josef, Michael Ellsworth, Miriam R.L. Petruck, Christopher R. Johnson and Jan Scheffczyk (2006) FrameNet II: Extended Theory and Practice. Available from the FrameNet website.

Russell, Betrand (1903) *Principles of Mathematics*, Cambridge University Press.

Sacks, H., E.A. Schegloff and G. Jefferson (1974) A simplest systematics for the organization of turn-taking for conversation, *Language*, Vol. 50, pp. 696–735.

Sag, Ivan A., Thomas Wasow and Emily M. Bender (2003) *Syntactic Theory: A Formal Introduction*, 2nd edition, CSLI Publications, Stanford.

de Saussure, Ferdinand (1916) *Cours de linguistique générale*, Payot, Lausanne and Paris, edited by Charles Bally and Albert Séchehaye.

Schlenker, Philippe (2011) Indexicality and *De Se* Reports, in C. Maienborn, K. v. Heusinger and P. Portner (eds.), *Semantics: an international handbook of natural language meaning*, pp. 1561–1604, de Gruyter.

Searle, John R. (1969) *Speech Acts: an Essay in the Philosophy of Language*, Cambridge University Press.

Seligman, Jerry and Larry Moss (1997) Situation Theory, in J. van Benthem and A. ter Meulen (eds.), *Handbook of Logic and Language*, North Holland and MIT Press.

Shanahan, Murray (2009) The Frame Problem, in E. N. Zalta (ed.), *The Stanford Encyclopedia of Philosophy*, winter 2009 edition, `http://plato.stanford.edu/archives/win2009/entries/frame-problem/`.

Shieber, Stuart (1986) *An Introduction to Unification-Based Approaches to Grammar*, CSLI Publications, Stanford.

Suppes, Patrick (1960) *Axiomatic Set Theory*, The University Series in Undergraduate Mathematics, D. van Nostrand Company, Inc.

Thomason, Richmond (1980) A model theory for propositional attitudes, *Linguistics and Philosophy*, Vol. 4, pp. 47–70.

Traum, David R. (1994) *A Computational Theory of Grounding in Natural Language Conversation*, PhD dissertation, University of Rochester, Department of Computer Science.

Turner, Raymond (2005) Semantics and Stratification, *Journal of Logic and Computation*, Vol. 15, No. 2, pp. 145–158.