Border Crossings

Jan van Eijck

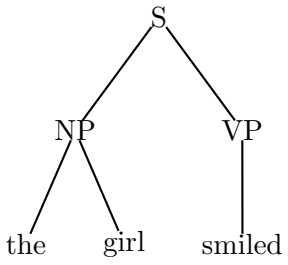
CWI and ILLC, Amsterdam, Uil-OTS, Utrecht

Abstract

It is well established by now that computer science has a number of concerns in common with natural language understanding. Common themes show up in particular with algorithmic aspects of text processing. This chapter gives an overview of border crossings from NLP to CS and back. Starting out from syntactic analysis, we trace our route via a philosophical puzzle about meaning, Hoare correctness rules for dynamic semantics, error state analysis of presupposition, equational reasoning about state change, programming with frameworks originally devised for natural language semantics, and the logic of incremental processing. In many cases methods and perspectives developed on one side of the border proved quite useful on the other side, although usually it becomes clear after a while that tools have to be redesigned to fit. The chapter ends with a sketch of the emerging contours of a 'proper treatment of context in natural language' that is more than just an application of a logic designed for program analysis.

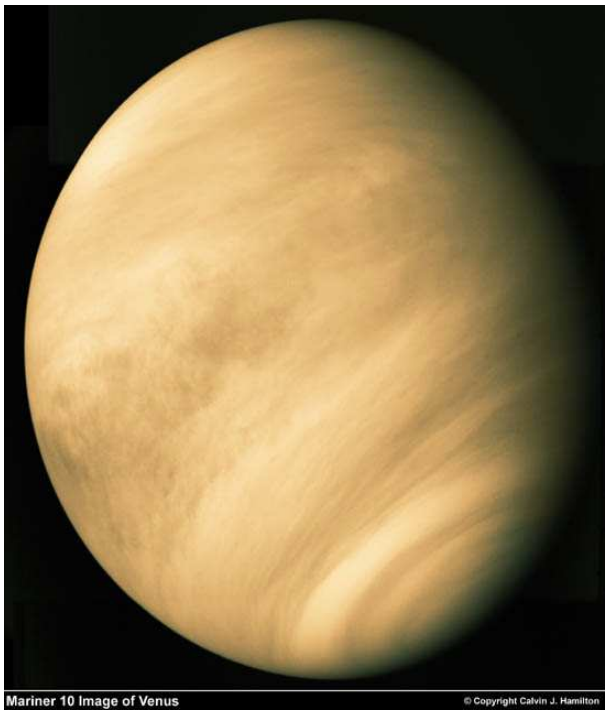
1 The borderline between natural languages and programming languages

One of the earliest examples of work on the borderline of natural language analysis and computer science is the mathematical theory of languages proposed by Noam Chomsky in the 1950s [7, 8]. Chomsky proposed to use context free rules as a grammar formalism: 'S $\longrightarrow$ NP VP' is an example of a rule saying that a sentence S consists of a nominal phrase NP followed by a verb phrase VP. Further rules of the same form then describe the internal structure of NP and VP. The context free rules can be used to assign syntax trees to sentences:



Chomsky and followers soon rejected this mathematical formalism as too weak for describing the rich structure of natural languages, but the theory of context free languages became part of the core of theoretical computer science [42]. Context free parsing is a well-understood subject and programming languages are often designed to be context free. Context free grammars made a glorious come-back with the advent of computational linguistics [51].

Figure 1: The planet Venus as seen by the Mariner 10 Spacecraft in 1974.



An old puzzle in the philosophy of language is the so-called paradox of the morning star, discussed already in [23]. If the phrase 'the morning star' and the phrase 'the evening star' both refer to Venus, then what kind of discovery did the (Babylonian?) astronomers make in Antiquity, when they found out that the morning star is identical to the evening star (Figure 2)? Frege ponders this, and concludes that there must be more to meaning than just reference. Whatever the Babylonians discovered, it was certainly not that Venus is identical to itself. The 'paradox of the morning star' is one of many puzzles of meaning. The chapters of Van Benthem and Dekker in this book list several others.

This puzzle of meaning turns up in a different guise in Montague grammar [44], a proposal to spell out the meanings of natural language sentences in a mathematically precise way. Montague

Figure 2: Venus Observational Parameters, from the NASA website.

Discoverer:	Unknown
Discovery Date:	Prehistoric
Distance from Earth	
Minimum (10^6 km)	38.2
Maximum (10^6 km)	261.0
Apparent diameter from Earth	
Maximum (seconds of arc)	66.0
Minimum (seconds of arc)	9.7
Maximum visual magnitude	-4.6
Mean values at inferior conjunction with Earth	
Distance from Earth (10^6 km)	41.44
Apparent diameter (seconds of arc)	60.2

distinguishes between extension and intension. The extension of 'the morning star' and 'the evening star' is the same; it is the planet Venus. But the situation might have been different. We might have lived in a world where 'the morning star' and 'the evening star' denote different planets. The intension of 'the morning star' is a map from possible worlds [40] to objects. Montague proposes an *intensional* logic to describe how the meanings (intensions) of complex expressions are built from the meanings (intensions) of their components.

In the 1970s, Van Emde Boas and Janssen drew attention to the parallel between Montague's intensions and assignments in programming [22]. The final chapter of Janssen's dissertation [33], a book presented as *an interdisciplinary study between mathematics, philosophy, computer science, logic and linguistics*, is a plea for the use of intensional logic in the analysis of assignment statements $x := t$ (assign to x the value t) in programming. Variable states in programming are very much like possible worlds in Montague-style natural language semantics.

The dynamic turn in natural language semantics, around 1980 [29, 35] started as a plea to take the context change potential of definite descriptions ('the morning star') and indefinite descriptions ('a farmer') seriously. Change of context is an action, and contexts are very much like variable states in programming. This time a border crossing in the other direction took place. Barwise [3] and Groenendijk and Stokhof [27] proposed to use a variation on a dynamic logic for program analysis [28] in a rational reconstruction of what it means to change context. Changing context is a change of information state, and, as Van Benthem's and Venema's chapters make clear, describing such state changes is the core business of dynamic logic.

2 Hoare Rules for Dynamic Semantics

One of the tried and trusted methods of formal program analysis is the method of correctness statements [31]. A Hoare correctness rule is a statement of the form $\{A\}\ S\ \{B\}$, where A, B are assertions about states, and S is a transition statement. The intended meaning is: if A is true in a state where S is executed, then B will be true in all of the result states. This kind of rule is used for the verification of programs. E.g., if A describes the state of my bank account at a certain date, and S is your action of transferring 1000 *euro* to my account, then B describes the improved state of my account after your action.

Dynamic semantics for natural language can be viewed as a plea for treating existential quantification as an action. Part of the meaning of *a farmer* is that a farmer gets introduced into the context. This action can be split up further as follows:

1. introduce an arbitrary individual,
2. check that that individual is a farmer.

The first of these two actions is the action of dynamic existential quantification. Appropriate states for this kind of action are variable assignments: maps from variables to objects in a suitable model. The action $\exists v$ is the action that resets the value of v to an arbitrary new value. Applying Hoare reasoning to this, we get, if we fix the postcondition B, that the so called weakest precondition A equals $\forall vB$:

$$\{\forall vB\}\ \exists v\ \{B\}.$$

Dynamic composition is treated in a way familiar from program analysis:

$$\frac{\{A\}\ S_1\ \{B\} \quad \{B\}\ S_2\ \{C\}}{\{A\}\ S_1; S_2\ \{C\}}$$

But if we want to use the format to describe the behaviour of the familiar negation-as-test (where $\neg S$, when executed in state s, succeeds and returns s in case S fails in s, and $\neg S$ fails in s in case S succeeds in s) we run into trouble. It turns out that the Hoare triple format does allow us to express 'from a state with property A there are no S transitions', (this is expressed by $\{A\}\ S\ \{\bot\}$), but not 'from a state with property A there are S transitions'. For example, $\{A\}\ S\ \{\top\}$ does not express this. Rather, it expresses something that always holds, for every state satisfies $\top$. The problem is connected to the well-known difficulty in computer science of capturing program termination in a Hoare style calculus [1].

One way of going about it is to introduce 'existential' Hoare triples to supplement the usual 'universal' Hoare triples. Read $(A)\ S\ (B)$ as: if A is true in a state, then execution of S in that state guarantees the existence of at least one result state satisfying B.

Now the behaviour of dynamic test negation is readily expressed in the extended Hoare format, as follows:

$$\frac{\{A\}\ S\ \{\bot\}}{(A)\ \neg S\ (A)} \qquad \frac{(A)\ S\ (\top)}{\{A\}\ \neg S\ \{\bot\}}$$

This was worked out as a sound and complete Hoare calculus for dynamic first order logic with definite descriptions and dynamic generalized quantifiers, in [19].

Still, the implicational Hoare format is a bit of a burden, and if one allows oneself the interweaving of state assertions and transition statements, why not go the whole way, and represent the state assertions as modalities in a version of quantified dynamic logic [28, 25]? As an example, the following axioms for quantified dynamic logic over DPL [27] appear in a sound and complete calculus presented in [12], very much in the style of the well-known Segerberg axiomatisation of propositional dynamic logic [50]:

$$\begin{aligned} [\exists v]A &\Leftrightarrow \forall v A \\ [\neg S]A &\Leftrightarrow ([S]\bot \Rightarrow A) \\ [Pt_1 \cdots t_n]A &\Leftrightarrow (Pt_1 \cdots t_n \Rightarrow A) \\ [S_1; S_2]A &\Leftrightarrow [S_1][S_2]A \end{aligned}$$

These axioms give in fact a translation instruction for DPL in the style of the rewriting rules for update formulas of Venema's chapter.

If we define dynamic implication $S_1 \Rightarrow S_2$ as $\neg(S_1; \neg S_2)$, we can use the calculus to compute the truth conditional meaning (as opposed to the 'context change potential') of sentence (1).

1 *If a linguist meets a computer scientist she ignores him.*

The translation of this sentence in Dynamic Predicate Logic is (2).

2 $(\exists x; Lx; \exists y; Cy; Mxy) \Rightarrow Ixy.$

The truth conditional meaning of (2) is given by:

$$\begin{aligned} \langle(\exists x; Lx; \exists y; Cy; Mxy) \Rightarrow Ixy\rangle\top &\Leftrightarrow [\exists x; Lx; \exists y; Cy; Mxy]\langle Ixy\rangle\top \\ &\Leftrightarrow [\exists x][Lx][\exists y][Cy][Mxy]\langle Ixy\rangle\top \\ &\Leftrightarrow \forall x(Lx \Rightarrow \forall y(Cy \Rightarrow (Mxy \Rightarrow \langle Ixy\rangle\top))) \\ &\Leftrightarrow \forall x(Lx \Rightarrow \forall y(Cy \Rightarrow (Mxy \Rightarrow Ixy))). \end{aligned}$$

3 Error Abortion and Presupposition

Error abortion in programming is modeled as a transition to a special error state ϵ in case in a state a programming error occurs, say a division by 0 is attempted:

$$\frac{s(z) = 0}{s \overset{y:=x/z}{\longrightarrow} \epsilon}$$

We can view error abortion as a 'third possibility' for how a dynamic statement may execute, in addition to success and failure. In a procedural view of natural language processing, this is easily accommodated, by introducing a distinguished state ϵ, and a means to talk about transitions to that state.

Another way to capture this 'third possibility' would be by distinguishing between S^+ transitions and S^- transitions, while making sure that $S^+ \cap S^- = \emptyset$. The S-transitions to ϵ from s can then be characterized as the following set:

$$\{(s, \epsilon) \mid \exists s' : \text{ not } s \xrightarrow{S^+} s' \text{ and not } s \xrightarrow{S^-} s'\}.$$

However, postulating an explicit error state allows one to do without the S^- transitions altogether (shaving them off with Occam's razor, so to speak).

Let us include ϵ among the states and make an agreement that everything holds at ϵ. So ϵ is an 'absurd state'. Call the ϵ state improper, all other states proper, and define two operators by means of:

- $[\![S]\!]A$, by definition true at a state in case every S-transition from that state leads to an A-state, i.e., to a proper A-state or to ϵ,
- $\langle\!\langle S \rangle\!\rangle A$, by definition true at a state in case some S-transition from that state leads to an A state, i.e., to a proper A-state or to ϵ.

Next, consider the meanings of $\neg[\![S]\!]\neg A$ and $\neg\langle\!\langle S \rangle\!\rangle \neg A$:

- $\neg[\![S]\!]\neg A$ is true at a state if not every S-transition from that state leads to ϵ or to a proper $\neg A$ state, i.e., if at least one S-transition from that state leads to a proper A state.
- $\neg\langle\!\langle S \rangle\!\rangle \neg A$ is true at a state if there is no S-transition from that state to ϵ or to a proper $\neg A$ state, i.e., if all S-transitions from that state are to proper A-states.

Abbreviate $\neg[\![S]\!]\neg A$ as $\langle S \rangle A$ and $\neg\langle\!\langle S \rangle\!\rangle \neg A$ as $[S]A$. Then it turns out that $[S]$ and $\langle S \rangle$ can be viewed as modalities corresponding to the transition relation $S \cap P^2$, where P is the set of proper states, while $[\![S]\!]$ and $\langle\!\langle S \rangle\!\rangle$ are interpreted as modalities corresponding to the transition relation S on the whole state set (including the improper state ϵ).

Also, it is easy to see that $\langle\!\langle S \rangle\!\rangle \bot$ is true in a state iff there is an S-transition to ϵ in that state. Thus, an alternative (and equivalent) set-up would be to dispense with $\langle\!\langle S \rangle\!\rangle$ and $[\![S]\!]$ altogether and to work only with dynamic modalities $\langle S \rangle$ and $[S]$ (but noting that they are not inter-definable), while adding a statement $a(S)$ for: there is an S-transition to ϵ from the current state. One should keep in mind, though, that $a(S)$ refers to a frame property.

Error abortion from a state s for a dynamic statement S takes place in case $\langle\!\langle S \rangle\!\rangle \bot \wedge [S]\bot$ is true at s. Thus, the abort condition (weakest precondition for error abortion, or presupposition failure) for a definite description that presupposes uniqueness of reference is:

$$\langle\!\langle \iota v : S \rangle\!\rangle \bot \wedge [\iota v : S]\bot.$$

There is a $\iota v : S$ transition to ϵ, and the execution of $\iota v : S$ does not lead to any proper states.

The weakest precondition for avoiding error abortion (i.e., the presupposition) of a definite description is captured by:

$$\langle \iota v : S \rangle \top \vee [\iota v : S]\bot.$$

Either execution of $\iota v : S$ succeeds (leads to a proper state), or it fails (leads to no states at all).

The weakest precondition for successful execution of 'the king is raging' (i.e., the condition for truth) is:

$$\begin{aligned} \langle \iota x : Kx; Rx \rangle \top &\Leftrightarrow \langle \iota x : Kx \rangle \langle Rx \rangle \top \\ &\Leftrightarrow \exists! x : Kx \wedge \exists x : (Kx \wedge \langle Rx \rangle \top) \\ &\Leftrightarrow \exists! x : Kx \wedge \exists x : (Kx \wedge Rx). \end{aligned}$$

The weakest precondition for failure of 'the king is raging' (i.e., the condition for falsity) is:

$$\begin{aligned} [\iota x : Kx; Rx]\bot &\Leftrightarrow [\iota x : Kx][Rx]\bot \\ &\Leftrightarrow \exists! x : Kx \wedge \forall x : (Kx \Rightarrow [Rx]\bot) \\ &\Leftrightarrow \exists! x : Kx \wedge \forall x : (Kx \Rightarrow \neg Rx). \end{aligned}$$

It follows from these two calculations that the presupposition of 'the king is raging' is the disjunction of these two formulas, i.e., as expected, the formula $\exists! x : Kx$.

This perspective on presupposition allows one to calculate presuppositions of complex expressions in terms of the meanings (not just the presuppositions!) of their parts. The error state semantics for presupposition takes left to right processing of text into account, so it arrives at plausible results for the presupposition of texts like (3).

3 *If France is a monarchy, then the Monarch of France is raging.*

The presupposition of (3) is $\top$, for the presupposition of the consequent 'the Monarch of France is raging' gets cancelled by the information conveyed by the antecedent 'France is a monarchy', and this is also what comes out in the error state semantics.

A full calculus for deriving presuppositions of complex sentences from the meanings of their components (plus their position in the text) in a way that fits the known data quite well can be built along these lines [11, 15, 13], which should be compared to the classical accounts in [24, 37, 38, 49]. Presupposition has everything to do with common grounds, the records that we assume to be public in any game of information exchange. Again, Dekker's chapter has more on this.

4 Equational Reasoning about State Change

What is the proper perspective to talk about states and state change in an abstract setting, say in the setting of typed logic? To see what a state is, we just have to notice what a state does, and model that. In the context of dynamic semantics (and imperative programming, for that matter), a state is a thing that enables the lookup of values of a set of stores or registers. For simplicity, let us assume that we have a set of registers R, and that these registers are used for storing values of type e. Then a register r is like a discourse referent in discourse representation theory [35], or a variable of dynamic predicate logic [27].

To talk about register assignment we need expressions $(r|E)$, where r is a register and E is an expression of type e. In a more general set-up we could introduce registers for other types, together with expressions $(r_T|E_T)$, for storing a new value of type T in a storage cell for that type. This would be suitable for a programming language with predefined types. We would have to be slightly careful with this, in case the types themselves involve states, for then the definition of states might become circular. For if the registers are of type T, the states must be the things that store values in these registers, i.e., the functions in $R \to D_T$, where D_T is the domain of things of type T (the things that can be stored). Call this set $D_\diamond$.

Now we know what the type of the expressions $(r|E)$ should be: $(\diamond, \diamond)$, for the act of putting a new value E in storage cell r effects a mapping from states to states. To avoid circularity in the definitions, we distinguish between standard types (types defined without the help of $\diamond$) and extended types (types construed by means of $\diamond$, possibly among other things). Storage cells are expressions of type $(\diamond, T)$, where T is a standard type.

If r is a register of type $(\diamond, T)$, then $[\![r]\!]$, the interpretation of r, is in the domain $D_\diamond \to D_T$, i.e., $[\![r]\!]$ is a map from states to values of the stored type. Assuming that storage cells can store items of all standard types, the domain $D_\diamond$ will consists of all those functions f from $\cup_T R_{\diamond T}$ to $\cup_T D_T$ with the property that $f(r) \in D_T$ iff r has type $(\diamond, T)$.

Given this architecture, we can give a set of equational rules for function application and state change. To the familiar rules that express reflexivity, symmetry and transitivity of equality, the rules of context that express that equalities may be applied inside expressions, and the beta axiom expressing that application terms are equal to the result of applying the function part to the argument part, and the eta axiom expressing extensionality, we add the following new ingredients:

Axioms for register lookup:

$$r_i((r_i|E)F) = E \qquad \frac{}{r_i((r_j|E)F) = r_iF}\ i \neq j$$

If you look up the contents of register r_i in a state that results from updating r_i in state F to value E, then you will find E.

If you look up the contents of register r_i in a state that results from updating r_j in state F to value E, then you can disregard the update of r_j, and just look up r_i in state F. This

expresses that the contents of store r_i does not depend on that of any of the other stores. The contents of the stores are independent.

Axioms for register update:

$$(r_i|E)((r_i|F)G) = (r_i|E)G \qquad \frac{}{(r_i|E)((r_j|F)G) = (r_j|F)((r_i|E)G)}\ i \neq j$$

The first one expresses that register update is destructive, the second one that order of update does not matter if the registers are different.

Finally, there is an axiom for register extensionality:

$$(r|(rE))E = E.$$

This expresses that updating r in state E with the value that it already has in that state (namely rE) changes nothing.

The theory sketched here and investigated more fully elsewhere can be viewed as the equational theory of destructive value assignment. The theory has a certain logical attraction. E.g., it turns out that universal validity for this calculus is decidable (though not of feasible complexity). The equational calculus can be extended to a full fledged dynamic typed logic, and be used for very simple and intuitive representations of dynamic semantics for natural language and denotational semantics for imperative programming [14]. The natural language application yields an elegant version of Dynamic Montague Grammar.

Still, after developing this system I have become convinced that destructive value assignment is a questionable ingredient in the representation of the process of dynamic context change in natural language. In retrospect, the development of typed logics with states should be viewed as more of a contribution to the semantics of imperative programming than to the semantics of natural language.

5 Programming with Dynamic First Order Logic

Indeed, I believe that the dynamic look at first order logic holds great promise as a programming paradigm. To make first order logic suitable for programming one must curb the force of quantification in some way or other. In logic programming, this is done by looking at universally quantified fragments only. The dynamic perspective on first order logic suggests that it could be done in other ways as well.

First, recall why dynamic first order logic is not suitable as a programming language *per se*. The instruction $\exists v$ is an instruction to replace the value of register v by an arbitrary new value. Computationally, the clause for quantification makes it computationally infeasible. For assume we are computing over an infinite domain, say the domain of natural numbers. Then $\exists v$ is an instruction to pick an arbitrary natural number and assign it to v. Since this can be done in an infinity of ways, this does not represent any finite computational procedure.

In the computational interpretation of dynamic first order logic we therefore change the quantifier action as follows. Instead of letting the quantifier action $\exists v$ perform its full duty, we split the action $\exists v$ in two tasks:

1. throwing away the old value of v, and
2. identifying appropriate new values for v.

In the absence of further information, any value for v is as good as another, so task (2) boils down to picking a random new object.

On infinite domains any attempt to perform task (2) will immediately cause an infinite branching transition. Computationally this is out, and therefore we have to *postpone* this task. We relegate the duty of finding an appropriate new value for v to an appropriate *identifying statement* for v further on. This procedure is familiar from constraint programming. A source of inspiration for this move, and thus for the computational interpretation of dynamic first order logic is *Alma-0*, a hybrid language for imperative programming mixed with logic programming developed by Apt c.s. [2].

Identifying statements for v are identity statements of the form $v = t$ or $t = v$. For computational purposes, states are partial functions $V \to D$, where V is the set of variables and D is the domain of the model that the computations are about. So suppose α is a state, and the computation process encounters $v = t$ in that state. If v^α is undefined, but t^α has a value, then we can use the statement $v = t$ to compute a value for v and extend the state to $\alpha' = \alpha \cup \{v/t^\alpha\}$. There are some annoying details to make sure that this process does not go wrong in the scope of a negation, but this is the basic picture. A further discussion of the executable interpretation of dynamic first order logic and of the *Dynamo* programming language based on it can be found in [16].

Of course, Dynamo computations will occasionally (in fact, quite often) have to yield the result 'I don't know', or more subtly, '. . . , and there may be further solutions', for dynamic first order logic is just as undecidable as standard first order logic. But the challenge is clear. The challenge is to extend the execution mechanism further and further in the direction of the (admittedly unattainable) ideal represented by the truth definition of dynamic first order logic.

Indeed, part of the attraction of dynamic logic programming is this presence of a logical ideal, in the form of the close connection with the definition of dynamic truth in first order logic. In extending the execution mechanism, we have a logical touch stone available: the executable interpretation has to remain faithful to the dynamic interpretation of first order logic in the following sense:

1. if the executable process interpretation computes an answer valuation, then that answer is correct according to the dynamic first order logic interpretation, and
2. if there are no answers according to the executable process interpretation then there are no answers according to the dynamic first order logic interpretation.

The present implementation of *Dynamo* is indeed faithful to dynamic first order logic. The extensions with constraint stacks that we are currently investigating have led to the development of a tableau calculus for dynamic first order logic [20].

6 The Dynamics of Context Change

Comparison of frameworks for natural language semantics is a notoriously difficult issue, especially if the frameworks are extremely expressive. Indeed, one would be hard put to come up with an example of a phenomenon that cannot be expressed in, say intensional typed logic. Proponents of Situation Semantics once made such claims with respect to the modeling of incomplete information (or 'partiality') in natural language semantics. Barwise even tried to argue that possible world semantics is incoherent [5], but was refuted (see, e.g., [45, 52]), as he himself admits in later work [4].

Still, comparison of frameworks for natural language semantics is possible, I claim, but one should be willing to take hints. As in metaphysics, hints that one is on a wrong track can be taken from the emergence of pseudo problems. And of course, as in metaphysics, one is never *forced* to take the hint.

In the case of natural language semantics, it seems reasonable to start out from certain basic facts about language processing — 'evidence that stares one in the face' — and ask *how naturally* a framework accounts for these. Thus, it seems to me that the starting point of an analysis of the dynamic process of context change in natural language should be a picture like the following:

$$[c_0, \ldots, c_{n-1}] \stackrel{\text{text}}{\longrightarrow} [c_0, \ldots, c_{n-1}] + [c_n, \ldots, c_k].$$

The process of interpreting a piece of text T presupposes a context of items — 'things in the real world' — that the text refers to. These items are used in the processing, indeed they are essential for making sense of the text. During the process of making sense of the text, the context is extended in turn with a list of topics introduced in T itself. This extended context then becomes available as a context for the processing of further text. This, in all its simplicity, is the basic picture.

There are several ways to model this, but destructive variable assignment is not one of them. The essential dynamics of dynamic first order logic resides in the interpretation of quantification as random assignment. This is borrowed from imperative programming, and, as we have seen, has a continuing appeal in the realm of programming. But is it intuitively the right feature for modeling the dynamics of context change, the process by which topics are added to the list of possible referents for pronouns in the course of a piece of natural language exchange? Destructive value assignment to a register destroys the previous value of that register. The fact that it is difficult to find a natural language counterpart for that process of destruction should make us pause.

If one uses dynamic variables destructively, the variables themselves become part of the semantic fabric, as we saw in Section 4, where states are things that have registers as essential ingredients. Thus, in picking a variable to introduce a new discourse referent, the question should be addressed *which* variable to use for that introduction. This of course depends on the variables already in use for the discourse referents that are 'part of the current context'. If there is no explicit representation of the current context, the problem remains elusive. In [21] an attempt was made to deal with it by devising strategies for 'merging' representations, but in retrospect I consider the 'merge problem' as a typical example of a pseudo problem caused by a wrong choice of primitives.

One way of avoiding the destructive assignment pseudo problem is by dispensing with dynamic variable *names* and handling argument binding in a way reminiscent of the so-called De Bruyn indices from lambda calculus [9]. This leads to a redesign of the dynamic logic paradigm for handling context and context change, as follows. In incremental dynamics [18], context gets represented as a stack of n objects. These contextually given objects need not all be different. Context gets processed dynamically: context extensions may be temporary. Of course, the anaphoric elements do not occur with an index in the text, as ANA_i; rather, the choice of appropriate indices for the anaphoric elements constitutes the process of anaphora resolution.

Text processing gets viewed as a process that adds a number of referents (say m) to the context: after processing the new text in a context of size n we have a new context of size $n+m$. Contextual elements that were mentioned too long ago may lose their salience (or: drop out of the context), but this process should be accounted for by an additional mechanism. Indeed, the incremental context change set-up sketches the general framework in which such a salience mechanism will have to fit.

4 *A man^i walked in. He$_i$ smiled.*

In textbooks on dynamic semantics one often finds pieces of example discourse like (4). The question *What do the indices in such example text mean?* has as its obvious answer: the indices do not belong the text *per se*, but are a shorthand to refer to one of the possible interpretations of the text in the given context. One of the possible interpretations of *he* in the second sentence, in the context established by the first sentence, is to link the pronoun to contextual item introduced by *a man*. The indices are just a gloss to indicate that this is how we suppose the anaphoric reference resolution mechanism links the pronoun from the second sentence to the noun phrase introduced in the first.

An anaphoric context is just a stack of n objects available as possible antecedents in future discourse. Introducing a new topic of conversation extends the anaphoric context by putting a new object on top of the context stack. The dynamic existential quantifier $\exists$ gets interpreted as the action of putting a new object on top of the context stack. If the size of the context is known, there is no need to indicate the register that gets bound by $\exists$. If the context is c, and its size is n, it is register c_n (on the assumption that we started counting context elements from

0).

A central idea is that the dynamic quantifier does not name the variable that it binds, but that dynamic quantification is always quantification in context. If a context is given, the interpretation of $\exists$ is just: introduce a next topic of conversation, and add it to the context. A system of dynamic reasoning without variables along these lines was proposed and analyzed in [18]. Compare also [10].

7 The Proper Treatment of Context in NL

A Montagovian treatment of the dynamics of context change should not start out from a first order logic with destructive assignment and extend this to a typed logic, like in Dynamic Montague Grammar [26], or the typed versions of discourse representation theory in [21, 39, 48], but from a process of context extension. What we need instead of a destructive Montague Grammar is a Contextual or Incremental Montague Grammar.

The Proper Treatment of Context for NL developed in [17] in terms of polymorphic type theory (see, e.g., [30, 43]) uses type specifications of contexts that carry information about the length of the context. E.g., the type of a context is given as $[e]_i$, where i is a type variable. Here, we will cavalierly use $[e]$ for the type of any context, and ι for the type of any index, thus relying on meta-context to make clear what the current constraints on context and indexing into context are. In types such as $\iota \rightarrow [e]$, we will tacitly assume that the index fits the size of the context. Thus, $\iota \rightarrow [e]$ is really a type scheme rather than a type, although the type polymorphism remains hidden from view. Since $\iota \rightarrow [e]$ generalizes over the size of the context, it is shorthand for the types $0 \rightarrow [e]_0$, $1 \rightarrow [e]_1$, $2 \rightarrow [e]_2$, and so on.

If $c :: [e]$ (i.e., c is a context) and $x :: e$ (x is an individual object), then we use $c\hat{}x$ for the context $c' :: [e]$ that is the result of appending x to c.

The translation of an indefinite noun phrase *a man* becomes something like:

5 $\lambda P \lambda c \lambda c'.\exists x(man\ x \wedge P|c|(c\hat{}x)c')$.

Here P is a variable of type $\iota \rightarrow [e] \rightarrow [e] \rightarrow t$, while c, c' are variables of type $[e]$ (variables ranging over contexts). The translation (5) has type $(\iota \rightarrow [e] \rightarrow [e]) \rightarrow [e] \rightarrow [e] \rightarrow t$. The P variable marks the slot for the VP interpretation. $|c|$ gives the length of the input context, i.e., the position of the next available slot. Note that $c\hat{}x[|c|] = x$.

Translation (5) does not introduce an anaphoric index, as in DPL based dynamic semantics for NL [26, 6, 32, 47, 48, 46, 14, 21, 39, 41]. All of these reconstructions are based in some way or other on a DPL-style treatment of register change [27], and they all inherit the main flaw of this approach: the destructive assignment problem. Interestingly, discourse representation theory itself did not suffer from this problem: the discourse representation construction algorithms of [35] and [36] are stated in terms of functions with finite domains, and carefully talk about

Figure 3: Code of an implementation of context logic for NL in Haskell.

```
type Context = [Entity]
type Prop = [Context]
type Trans = Context -> Prop
type Idx = Int

neg :: Trans -> Trans
neg = \ phi c -> if phi c == [] then [c] else []
conj :: Trans -> Trans -> Trans
conj = \ phi psi c -> concat [ psi c' | c' <- (phi c) ]
impl :: Trans -> Trans -> Trans
impl = \ phi psi ->  neg (phi `conj` (neg psi))
exists :: Trans
exists = \ c -> [ (extend c x) | x <- [minBound..maxBound]]
forall :: Trans -> Trans
forall = \ phi -> neg (exists `conj` (neg phi))
self :: (a -> a -> b) -> a -> b
self = \ p x -> p x x

intS :: S -> Trans
intS (S np vp) = (intNP np) (intVP vp)
intS (If s1 s2) = (intS s1) `impl` (intS s2)

intNP :: NP -> (Idx -> Trans) -> Trans
intNP (NP1 det cn) = (intDET det) (intCN cn)

intVP :: VP -> Idx -> Trans
intVP (VP1 tv np) = \ subj -> intNP np (\ obj -> intTV tv obj subj)
intVP (VP2 tv _) = self (intTV tv)

intDET Some = \ phi psi c -> let i = length c in
                 (exists `conj` (phi i) `conj` (psi i)) c
intDET Every = \ phi psi c -> let i = length c in
                 neg (exists `conj` (phi i) `conj` (neg (psi i))) c
intDET No = \ phi psi c -> let i = length c in
                 neg (exists `conj` (phi i) `conj` (psi i)) c
intDET The = \ phi psi c -> let i = length c in
                 ((unique i (phi i)) `conj`
                    exists `conj` (phi i) `conj` (psi i)) c
```

'taking a fresh discourse referent' to extend the domain of a verifying function, for each new noun phrase to be processed.

Instead, an anaphoric index i is picked up from the input context. Also, the context is not reset but incremented: context update is not destructive like in DPL.

Here are appropriate dynamic operations in typed logic. Assume ϕ and ψ have the type of context transitions, i.e., type $[e] \to [e] \to t$, and that c, c', c'' have type $[e]$. Note that $\hat{}$ is an operation of type $[e] \to e \to [e]$. Then we get:

$$\begin{aligned} \exists &:= \lambda cc'.\exists x(c\hat{}x = c') \\ \neg\phi &:= \lambda cc'.(c = c' \wedge \neg\exists c''\phi cc'') \\ \phi;\psi &:= \lambda cc'.\exists c''(\phi cc'' \wedge \psi c''c') \end{aligned}$$

These operations encode the semantics for incremental quantification, dynamic incremental negation and dynamic incremental conjunction in typed logic. Figure 3 gives part of the code of an implementation of a system along these lines in a state of the art functional programming language, Haskell [34]. Building such systems serves at least two purposes. By constructing systems that can do things that humans do we gain insight in what it is that humans can do. More practically, algorithms that can resolve reference of pronouns in a text are crucial for the task of information extraction from natural language texts that is a main topic of De Rijke's contribution to this book.

8 Context Semantics and Reference Resolution

Pronoun resolution should resolve pronouns to the most salient referent in context, modulo additional constraints such as gender agreement (the referent should have the same gender as the pronoun).

To handle salience, we need contexts with more structure, so that context elements can be permuted without danger of losing track of them. Contexts as lists of elements under a permutation are conveniently represented as lists of index/element pairs. To talk about the permutation **bcad** of the context **abcd** we represent **abcd** as $[(0, \mathbf{a}), (1, \mathbf{b}), (2, \mathbf{c}), (3, \mathbf{d})]$ and reshuffle this to $[(2, \mathbf{c}), (1, \mathbf{b}), (0, \mathbf{a}), (3, \mathbf{d})]$. Then we can continue to use index 2 to pick up the reference to **c**, no matter where **c** ends up in the reshuffled context, while the list ordering encodes salience of the items. Figure 4 gives code for reference resolution in such contexts.

6 *He loved some woman.*

In a context where referents for the pronoun are available, *he* in example (6) can be resolved to any referent that satisfies the property. Suppose our initial context is $[(3, A), (2, M), (1, B), (0, J)]$, where *Bill* and *John* are men, and *Ann* and *Mary* are women. When evaluating the example sentence in this context, in a situation where only B is a lover, this is what we get:

Figure 4: Code of an implementation of reference resolution in context logic.

```
resolveMASC :: Context' -> [Idx]
resolveMASC (c,co)  = resolveMASC' c where
  resolveMASC' []                      = []
  resolveMASC' ((i,x):xs) | man x      = i : resolveMASC' xs
                          | otherwise = resolveMASC' xs
resolveFEM :: Context' -> [Idx]
resolveFEM (c,co)  = resolveFEM' c where
  resolveFEM' []                     = []
  resolveFEM' ((i,x):xs) | woman x   = i : resolveFEM' xs
                         | otherwise = resolveFEM' xs
resolveNEUTR :: Context' -> [Idx]
resolveNEUTR (c,co)  = resolveNEUTR' c where
  resolveNEUTR' []                      = []
  resolveNEUTR' ((i,x):xs) | thing x   = i : resolveNEUTR' xs
                           | otherwise = resolveNEUTR' xs
```

```
CS> eval' example6
[([(1,B),(4,A),(3,A),(2,M),(0,J)],[Loved 1 4]),
 ([(1,B),(4,M),(3,A),(2,M),(0,J)],[Loved 1 4])]
CS>
```

Since only B is a lover, *he* gets resolved to B, and the new contexts have B and Bs loved one as the two most salient items, with the subject more salient than the object.

7 *He hated some thing.*

In example (7) we expect that we get all the new contexts where *Bill* or *John* with their objects of hatred are added, again with the subjects more salient than the objects. And this is what we get:

```
CS> eval' example7
[([(1,B),(4,E),(3,A),(2,M),(0,J)],[Hated 1 4]),
 ([(1,B),(4,F),(3,A),(2,M),(0,J)],[Hated 1 4]),
 ([(1,B),(4,G),(3,A),(2,M),(0,J)],[Hated 1 4]),
 ([(1,B),(4,H),(3,A),(2,M),(0,J)],[Hated 1 4]),
 ([(1,B),(4,I),(3,A),(2,M),(0,J)],[Hated 1 4]),
 ([(1,B),(4,K),(3,A),(2,M),(0,J)],[Hated 1 4]),
```

```
 ([(1,B),(4,L),(3,A),(2,M),(0,J)],[Hated 1 4]),
 ([(0,J),(4,E),(3,A),(2,M),(1,B)],[Hated 0 4]),
 ([(0,J),(4,F),(3,A),(2,M),(1,B)],[Hated 0 4]),
 ([(0,J),(4,G),(3,A),(2,M),(1,B)],[Hated 0 4]),
 ([(0,J),(4,H),(3,A),(2,M),(1,B)],[Hated 0 4]),
 ([(0,J),(4,I),(3,A),(2,M),(1,B)],[Hated 0 4]),
 ([(0,J),(4,K),(3,A),(2,M),(1,B)],[Hated 0 4]),
 ([(0,J),(4,L),(3,A),(2,M),(1,B)],[Hated 0 4])]
CS>
```

Reference resolution is crucial for natural language understanding, and algorithms like the one sketched here will no doubt play a key part in tools for information extraction from natural language text.

9 Conclusion

Border crossings between natural language analysis and analysis of programming, and between logic and other neighbouring disciplines like cognitive science, game analysis, occur all the time. That our community is at ease in these areas where various disciplines touch is also made clear by the other contributions to this volume. This was just a selection of border crossings in which some of us were involved.

Working on the borderline between several areas it is sometimes hard to decide what tools to use. It is quite tempting to select a nice method from computer science or language analysis, and put it to work on the other side of the border. Grab a hammer, and everything looks like a nail. After an initial stage of enthousiasm, awareness dawns, usually, that some of the tools have to be redesigned.

It may happen that when engaged in research in an application area of logic, one crosses borders while remaining unaware of the fact. It is only in retrospect that it becomes clear on which side of the border one has been at work. But then, sometimes this does not matter, and it may even add to the fun.

References

[1] K.R. Apt. Ten years of Hoare's logic: A survey—part i. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, 1981.

[2] K.R. Apt, J. Brunekreef, V. Partington, and A. Schaerf. Alma-0: An imperative language that supports declarative programming. *ACM Toplas*, 20:1014–1066, 1998.

[3] J. Barwise. Noun phrases, generalized quantifiers and anaphora. In P. Gärdenfors, editor, *Generalized Quantifiers: linguistic and logical approaches*, pages 1–30. Reidel, Dordrecht, 1987.

[4] J. Barwise. *The Situation in Logic*. CSLI Lecture Notes. University of Chicago Press, 1989.

[5] J. Barwise and J. Perry. *Situations and Attitudes*. MIT Press, Cambridge Mass, Cambridge, MA, 1983.

[6] G. Chierchia. Anaphora and dynamic binding. *Linguistics and Philosophy*, 15(2):111–183, 1992.

[7] N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, 1956.

[8] N. Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2):137–167, 1959.

[9] N.G. de Bruijn. A survey of the project AUTOMATH. In J.R. Hindley and J.P. Seldin, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 579–606. Academic Press, London, 1980.

[10] P. Dekker. Predicate logic with anaphora. In L. Santelmann and M. Harvey, editors, *Proceedings of the Fourth Semantics and Linguistic Theory Conference*, page 17 vv, Cornell University, 1994. DMML Publications.

[11] J. van Eijck. The dynamics of description. *Journal of Semantics*, 10:239–267, 1993.

[12] J. van Eijck. Axiomatizing dynamic predicate logic with quantified dynamic logic. In J. van Eijck and A. Visser, editors, *Logic and Information Flow*, pages 30–48. MIT Press, Cambridge Mass, 1994.

[13] J. van Eijck. Presuppositions and information updating. In H. de Swart M. Kanazawa, C. Piñon, editor, *Quantifiers, Deduction, and Context*, pages 87–110. CSLI, Stanford, 1996.

[14] J. van Eijck. Typed logics with states. *Logic Journal of the IGPL*, 5(5):623–645, 1997.

[15] J. van Eijck. Praktische filosofie (boekbespreking). *ANTW*, 1998.

[16] J. van Eijck. Programming with dynamic predicate logic. Technical Report CT-1998-06, ILLC, 1998. Available from `www.cwi.nl/~jve/dynamo`.

[17] J. van Eijck. The proper treatment of context in NL. In Paola Monachesi, editor, *Computational Linguistics in the Netherlands 1999; Selected Papers from the Tenth CLIN Meeting*, pages 41–51. Utrecht Institute of Linguistics OTS, 2000.

[18] J. van Eijck. Incremental dynamics. *Journal of Logic, Language and Information*, 10:319–351, 2001.

[19] J. van Eijck and F.J. de Vries. Dynamic interpretation and Hoare deduction. *Journal of Logic, Language, and Information*, 1:1–44, 1992.

[20] J. van Eijck, J. Heguiabehere, and B. Ó Nualláin. Tableau reasoning and programming with dynamic first order logic. *Logic Journal of the IGPL*, 9(3), May 2001.

[21] J. van Eijck and H. Kamp. Representing discourse in context. In J. van Benthem and A. ter Meulen, editors, *Handbook of Logic and Language*, pages 179–237. Elsevier, Amsterdam, 1997.

[22] P. van Emde Boas and T. Janssen. Montague grammar and programming languages. In J Groenendijk and M. Stokhof, editors, *Proceedings of the Second Amsterdam Colloquium*, pages 101–124, Philosophy Department, University of Amsterdam, 1978.

[23] G. Frege. Ueber sinn und bedeutung. Translated as 'On Sense and Reference' in Geach and Black (eds.), Translations from the Philosophical Writings of Gottlob Frege, Blackwell, Oxford (1952), 1892.

[24] G. Gazdar. A solution to the projection problem. In C.-K. Oh and D. Dinneen, editors, *Syntax and Semantics 11: Presupposition*, pages 57–89. Academic Press, New York, 1979.

[25] R. Goldblatt. *Axiomatising the Logic of Computer Programming.* Springer, 1982.

[26] J. Groenendijk and M. Stokhof. Dynamic Montague Grammar. In L. Kalman and L. Polos, editors, *Papers from the Second Symposium on Logic and Language*, pages 3–48. Akademiai Kiadoo, Budapest, 1990.

[27] J. Groenendijk and M. Stokhof. Dynamic predicate logic. *Linguistics and Philosophy*, 14:39–100, 1991.

[28] D. Harel. *First-Order Dynamic Logic.* Number 68 in Lecture Notes in Computer Science. Springer, 1979.

[29] I. Heim. *The Semantics of Definite and Indefinite Noun Phrases.* PhD thesis, University of Massachusetts, Amherst, 1982.

[30] J. Roger Hindley. *Basic Simple Type Theory.* Cambridge University Press, 1997.

[31] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):567–580, 583, 1969.

[32] Martin Jansche. Dynamic Montague Grammar lite. Dept of Linguistics, Ohio State University, November 1998.

[33] T.M.V. Janssen. *Foundations and Applications of Montague Grammar*. CWI Tract 19. CWI, Amsterdam, 1986.

[34] S. Peyton Jones, J. Hughes, et al. Report on the programming language Haskell 98. Available from the Haskell homepage: `http://www.haskell.org`, 1999.

[35] H. Kamp. A theory of truth and semantic representation. In J. Groenendijk et al., editors, *Formal Methods in the Study of Language*. Mathematisch Centrum, Amsterdam, 1981.

[36] H. Kamp and U. Reyle. *From Discourse to Logic*. Kluwer, Dordrecht, 1993.

[37] L. Karttunen. Presuppositions of compound sentences. *Linguistic Inquiry*, 4:169–193, 1973.

[38] L. Karttunen and S. Peters. Conventional implicature. In C.-K. Oh and D. Dinneen, editors, *Syntax and Semantics 11: Presupposition*, pages 1–56. Academic Press, 1979.

[39] M. Kohlhase, S. Kuschert, and M. Pinkal. A type-theoretic semantics for λ-DRT. In P. Dekker and M. Stokhof, editors, *Proceedings of the Tenth Amsterdam Colloquium*, Amsterdam, 1996. ILLC.

[40] S.A. Kripke. Naming and necessity. In D. Davidson and G. Harman, editors, *Semantics of Natural Language*, pages 253–355. Reidel, Dordrecht, 1972.

[41] S. Kuschert. *Dynamic Meaning and Accommodation*. PhD thesis, Universität des Saarlandes, 2000. Thesis defended in 1999.

[42] H.R. Lewis and C.H. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, 1981.

[43] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17, 1978.

[44] R. Montague. The proper treatment of quantification in ordinary English. In J. Hintikka e.a., editor, *Approaches to Natural Language*, pages 221–242. Reidel, 1973.

[45] R. Muskens. A relational formulation of the theory of types. *Linguistics and Philosophy*, 12:325–346, 1989.

[46] R. Muskens. A compositional discourse representation theory. In P. Dekker and M. Stokhof, editors, *Proceedings 9th Amsterdam Colloquium*, pages 467–486. ILLC, Amsterdam, 1994.

[47] R. Muskens. Tense and the logic of change. In U. Egli et al., editor, *Lexical Knowledge in the Organization of Language*, pages 147–183. W. Benjamins, 1995.

[48] R. Muskens. Combining Montague Semantics and Discourse Representation. *Linguistics and Philosophy*, 19:143–186, 1996.

[49] S. Peters. A truth-conditional formulation of Karttunen's account of presupposition. *Texas Linguistic Forum*, pages 137–149, 1977.

[50] K. Segerberg. A completeness theorem in the modal logic of programs. In T. Traczyck, editor, *Universal Algebra and Applications*, pages 36–46. Polish Science Publications, 1982.

[51] K. Sikkel. *Parsing Schemata – A Framework for Specification and Analysis of Parsing Algorithms*. Springer, 1997.

[52] R. Stalnaker. Possible worlds and situations. *Journal of Philosophical Logic*, 15:109–123, 1986.