A strategy for dynamic interpretation: a fragment and an implementation

O. Bouchez, D.J.N. van Eijck, O. Istace

Computer Science/Department of Software Technology

Report CS-R9317 March 1993

CWI is the National Research Institute for Mathematics and Computer Science. CWI is part of the Stichting Mathematisch Centrum (SMC), the Dutch foundation for promotion of mathematics and computer science and their applications. SMC is sponsored by the Netherlands Organization for Scientific Research (NWO). CWI is a member of ERCIM, the European Research Consortium for Informatics and Mathematics.

P.O. Box 4079, 1009 AB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

A Strategy for Dynamic Interpretation: a Fragment and an Implementation

Olivier Bouchez[1,2], Jan van Eijck[2,3] and Olivier Istace[1,2]

EMAIL: obo@info.fundp.ac.be, jve@cwi.nl, ois@info.fundp.ac.be

[1]Institut d' Informatique, FUNDP, 61 Rue de Bruxelles, 5000 Namur, Belgium,

[2]CWI, P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

[3]OTS, Trans 10, 3512 JK Utrecht, The Netherlands

Abstract

The strategy for natural language interpretation presented in this paper implements the dynamics of context change by translating natural language texts into a meaning representation language consisting of (descriptions of) programs, in the spirit of dynamic predicate logic (DPL) [5]. The difference with DPL is that the usual DPL semantics is replaced by an error state semantics [2]. This allows for the treatment of unbound anaphors, as in DPL, but also of presuppositions and presupposition projection.

The use of this dynamic interpretation strategy is demonstrated in an implementation of a small fragment of natural language which handles unbound pronoun antecedent links, where it is assumed that the intended links are indicated in the input string, and uniqueness presuppositions of definite descriptions. The implementation consists of a syntax module which outputs parse trees, a semantic module mapping parse trees to DPL representations, a representation processor which determines truth conditions, falsity conditions and presupposition failure conditions, and an evaluator of these conditions in a database model.

The implementation uses the logic programming language Gödel [6], an experimental successor of Prolog, with similar functionality and expressiveness, but with an improved declarative semantics.

1991 CR Subject Classification: I.2.1, I.2.7

Keywords and Phrases: natural language semantics, dynamic interpretation, presupposition.

Note: This paper will appear in the Proceedings of the Sixth Conference of the European Chapter of the Association for Computational Linguistics, 21–23 April 1993, Utrecht.

Report CS-R9317
ISSN 0169-118X
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

1 The Idea of Dynamic Interpretation

Recent developments in Natural Language semantics have witnessed a shift away from static representation languages towards representation languages with a dynamic flavour. Such representation languages can be viewed as definitions of very simple imperative programming languages.

To see how the imperative style comes in, consider the treatment of indefinite descriptions (or: existential phrases). Existential quantifiers are viewed dynamically as random assignment statements followed by tests. The translation of the natural language phrase 'a man' becomes something like:

```
x := ?; man(x)
```

The first part of this statement can be viewed as a random assignment to register x, the second part as a test on the value of x. This sequence of instructions is performed against the background of a database, i.e., a model of first order logic. The sequence succeeds if the database contains (representations of) men, and it can succeed in as many ways as there are men available in the database.

The motivating examples for the shift from static to dynamic representation have to do with pronoun binding. The translation of phrases like 'a man' in terms of assignments of values to registers makes it possible to treat binding of pronouns across sentence boundaries (the next sentence can start with 'He' to pick up the reference to 'a man'). The nice thing about the treatment in terms of assignment is that the scope of the existential quantification is not closed off at the end of a sentence, as used to be the case for NL systems that employ static representation (in terms of the existential quantifiers of predicate logic, with their irritating closing brackets).

Recently, it has become clear that dynamic representation has some other interesting features:

- It becomes possible to give an account of presupposition failure phenomena in terms of the definition of an error state semantics for the dynamic representation language [3, 2]. Presupposition failure occurs for example if one tries to interpret "John's wife is unhappy" in a situation where John is not married.
- A more natural treatment of tense becomes possible. A sequence of sentences in the past tense like "A man walked in. He sat down. He ordered a drink" etc, is represented using subsequent assignments of values (time intervals) to a dedicated time register t [10].

The dynamic representation language can be analysed with tools that were originally designed for analysing imperative programming languages, namely the tools for precondition reasoning from Hoare logic or dynamic logic [11]. Precondition reasoning for dynamic predicate logic with standard semantics was introduced in [4]. Precondition reasoning gives the truth conditions of DPL representations in the form of formulas of first order logic (FOL). When applied to the error state semantics of DPL, precondition reasoning can also be used to find the presupposition failure conditions of DPL representations as FOL formulas.

We provide an integrated treatment of syntax and semantics of a small fragment of natural language and test this by implementing it. The syntax of our toy grammar is a version of categorial grammar with feature unification. The semantics uses DPL representations, with an error state semantics which is reflected in the rules for precondition reasoning implemented in the precondition module. This module generates predicate logical formulas expressing the weakest preconditions of success, failure or error of the DPL representations.

In detail, our interpretation strategy consists of the following steps:

1. Parsing a sentence or text and building a representation tree of its structure.
2. Translating the parse tree into a DPL program.
3. Using precondition reasoning to compute preconditions as formulas of FOL.
4. Simplifying the preconditions using a simplifier for FOL formulas.
5. Evaluating the resulting formulas in a database model.

The current implementation produces for an input text within the grammar fragment: a LaTeX form report containing the sentence, the parse tree, the DPL translation, the precondition of success, the precondition of failure and the precondition of error, all in simplified form, and the result of evaluation in the database.

2 Dynamic Predicate Logic

2.1 Informal discussion

DPL meaning representations for natural language sentences can be viewed as procedures or programs with a relational semantics. The programs that represent the meanings are interpreted as relations between input states and output states. A state is a mapping from variables to values in a model (in our simple set-up all variables are of the same type). The representation for an example sentence such as "John saw a man" is a program which associates *John* with a variable x, *a man* with a variable y, and first checks whether the value of x equals John, next puts a value in y which satisfies the predicate of being a man, and finally checks whether the values of x and y are such that the first saw the second.

Thus, the representation of "John saw a man" is a program which relates input states where x is mapped to John to output states where x is mapped to John and y is mapped to some man seen by John. If the evaluation takes place in a model where John saw several men, then there are several possible output states. If the evaluation takes place in a model where John saw no men at all, then there is no output. A program that yields no output for a given input fails for that input. A program yielding at least one output for a given input succeeds for that input. A program which yields at most one output for a given input is deterministic for that input. A program which yields more than one output for a given input is indeterministic for that input. The example "John saw a man" shows that indefinite descriptions may give rise to indeterministic programs. Deterministic programs that do not change their input are called test

programs. If a test program succeeds, its output equals its input. The sentence "John saw him" would give rise to a test program. Assuming that the variable x, y are used for the subject and object of the sentence, respectively, the program will succeed for any input with x mapped to John and y mapped to some male individual seen by John. In this case success means that the output state equals the input state. The program will fail for any other input.

All basic programs of DPL are tests; they do not change their input, and they succeed if the values of terms are in a specified relation and fail otherwise.

Indeterminism in DPL arises from assignment programs. The assignment program for an indefinite description *a man* will assign a new value to a variable x and succeed for any value of x which is a man. This is called indefinite assignment. The assignment program for a definite description *the manager* gives a value to a variable if and only if there is only one possible value in the model under consideration.

Complex programs can be formed by means of negation, implication and sequential composition. Negation and implication always form tests, but sequential composition does not. Sequential compositions are tests if and only if the component programs are tests.

2.2 Syntax

For ease of exposition we will assume there are no function symbols in the DPL representation language, so the terms of DPL are either constants or variables. Let C be the set of constants, V the set of variables, and assume $c \in C, v \in V$.

DPL terms $t ::= c \mid v$.

Assume a set of relation symbols R with arities. Then the programs of DPL are given by the following BNF definition.

DPL programs $\pi ::= t = t \mid Rt \cdots t \mid (\pi;\pi) \mid (\pi \Rightarrow \pi) \mid (\neg\pi) \mid \eta v : \pi \mid \iota v : \pi$.

We will use *man*, *see* as the relation symbols that translate "man", "see", and so on. Thus, (1) is a DPL program.

(1) $\quad (\eta v_2 : man(v_2); see(v_2, v_4))$.

We will omit outermost brackets and brackets in sequential compositions like $((\pi_1;\pi_2);\pi_3)$. This is harmless, for sequential composition is associative. Also, we will abbreviate $\eta v : v = t$ as $v := t$. This abbreviation is natural, as the sequential composition of random assignment to v and test for equality with t boils down to assigning the value of t to v.

2.3 Indices for Antecedents and Anaphors

In the natural language fragment we treat, we use co-indexing to indicate intended anaphoric links. We follow Barwise [1] in using superscripts for antecedents and subscripts for anaphors.

(2) $\quad$ *A man walked in. He smiled.*

If we intend the pronoun in (2) to refer to the subject of the first sentence, we indicate this intention as follows.

(3) *A man^1 walked in. He$_1$ smiled.*

The superscript on the indefinite noun phrase indicates that this NP acts as an antecedent for NPs with the same index as a subscript. The subscript on the pronoun indicates the antecedent to which the pronoun is linked.

The use of subscripts and superscripts is necessary because noun phrases can act as anaphors and antecedents at the same time.

(4) *A man^1 walked in.*
Another man$^2_1$ walked out.
He$_2$ was angry.

In example (4) the noun phrase *another man* is anaphorically constrained by an antecedent noun phrase *a man* (it must have a different referent), and at the same time acts as antecedent for the second occurrence of *a man.*

The superscripts and subscripts refer to the variables we employ in the translation of the noun phrases. Superscripts correspond to variables that get assigned a value in the translation, subscripts to variables that are simply used. Sentence (5) will get translated as (6) (tense is ignored, here and hereafter, for ease of exposition).

(5) *John^1 saw a man^2.*

(6) $v_1 := J; \eta v_2 : man(v_2); see(v_1, v_2).$

Sentence (7) gets translated as (8).

(7) *Mary^3 ignored him$_1$.*

(8) $v_3 := M; ignore(v_3, v_1).$

Sentence (9) gets translated as (10).

(9) *She$_3$ saw another man$^4_2$.*

(10) $\eta v_4; v_4 \neq v_2; man(v_4); see(v_3, v_4).$

Turning now to definite descriptions, the natural translation of example (11) is (12).

(11) *John^1 saw the man^2.*

(12) $v_1 := J; \iota v_2 : man(v_2); see(v_1, v_2).$

In the error state semantics for DPL that we have in mind for this, (12) gives error in every model where there is no unique man. It is clear that in most cases this is too strong. Still, we do not think this is a serious problem for our general approach. It seems to be a linguistic fact that definite descriptions often are used in a context-dependent way, to designate a unique referent in a very specific context, which however is not made fully explicit.

One context where (11) makes perfect sense is a situation where John and some other male individual are present, and where it is left implicit that John is excluded from the context where the reference is unique. In such cases we propose to read the definite description as uniquely satisfying the description plus the extra condition of being non-identical with some constraining

antecedent, in this case the subject of the sentence. This strategy boils down to reading (11) as (13).

(13) *John*^{1} *saw the other man*$^2_1$.

Here the determiner *the other*$_i$ is treated similarly to *another*$_i$. This gives translation (14).

(14) $v_1 := J; \iota v_2 : (v_2 \neq v_1; man(v_2));$
$see(v_1, v_2).$

In many cases another mechanism seems to be at work.

(15) *A man walked in. John saw the man.*

Example (15) has a natural reading where the definite description is anaphorically linked to an antecedent. We propose to make such implicit anaphoric links explicit, as in (16).

(16) *A man*^{1} *walked in. John*^{2} *saw the man*$^3_1$.

If we provide the right translation instruction for such anaphoric uses of *the*, we arrive at translation (17).

(17) $\eta v_1 : man(v_1); walk\text{-}in(v_1); v_2 := J;$
$\iota v_3 : (v_3 = v_1; man(v_3)); see(v_2, v_3).$

This gives *the man*$^3_1$ the meaning: *the unique man that is equal to* v_1, with v_3 available for later reference to this individual. It seems to us that this gives the correct result, in the present case and in lots of other cases.

In the case of (18) we still run into trouble, however.

(18) *The man with the hat smiled.*

Here, the natural translation is (19).

(19) $\iota v_1 : (man(v_1); \iota v_2 : hat(v_2); is\text{-}of(v_2, v_1);$
$smile(v_1).$

This translation contains a definite assignment $\iota v_2 : hat(v_2)$, so it seems to assume that there is a unique hat in the domain of discourse, which is perhaps a bit too strong. There are at least the following two ways out. One is by handwaving. Just remark that in descriptions like *the man with the golden gun*, the second definite article is not quite as definite as it looks, and the description is in fact idiomatic for the more strictly correct *the man with a golden gun*. The other escape is to add an epicycle to the analysis, in order to achieve that *man*^{1} *with*$_1$ *the hat*^{2} translates into (20).

(20) $man(v_1); \iota v_2 : (hat(v_2); is\text{-}of(v_2, v_1)).$

We provisionally opt for the first solution.

2.4 Semantics

The standard DPL semantics maps input states to sets of possible output states. Let a model $\mathcal{M} = \langle M, I \rangle$, where M is the domain and I the interpretation function for a set of constants and

relation symbols be given. Then the set of states is the set of functions M^V, and the standard semantics for DPL is given by a function $[\![\cdot]\!]_{\mathcal{M}} : M^V \to \mathcal{P}(M^V)$.

In order to capture the uniqueness presuppositions of definite descriptions, we replace the standard semantics by an error state semantics. In a Russellian account of definite descriptions, "The king of France is bald" when evaluated with respect to the state of affairs in 1905 or 1993 is false, for there is no unique referent for the description. But it is much more natural to follow Frege, Strawson and the majority of the linguistic community in assuming that statements involving "the king of France", when interpreted with respect to a state of affairs where there is no unique king of France, may be neither true nor false, because they suffer from presupposition failure.

We propose to use an error state semantics to take in account the failure of uniqueness presuppositions of ι assignments. The error state semantics of DPL if given by a function

$$[\![\cdot]\!]_{\mathcal{M}} : (M^V \cup \epsilon) \to \mathcal{P}(M^V \cup \epsilon).$$

In the definition of this function, which follows, ϵ refers to a special error state, A ranges over proper states, B ranges over states in general (including the error state), and $A[x := d]$ is used for the proper state which is like A, except for the fact that x is mapped to d.

1. $[\![\pi]\!]_{\mathcal{M}}(\epsilon) = \{\epsilon\}$
2. $[\![Rt_1 \cdots t_n]\!]_{\mathcal{M}}(A) =$

$$\begin{cases} \{A\} & \text{if } \langle V_{\mathcal{M},A}(t_1), \ldots, V_{\mathcal{M},A}(t_n)\rangle \in I(R) \\ \emptyset & \text{otherwise.} \end{cases}$$

3. $[\![t_1 = t_2]\!]_{\mathcal{M}}(A) =$

$$\begin{cases} \{A\} & \text{if } V_{\mathcal{M},A}(t_1) = V_{\mathcal{M},A}(t_2) \\ \emptyset & \text{otherwise.} \end{cases}$$

4. $[\![(\pi_1;\pi_2)]\!]_{\mathcal{M}}(A) =$

$\bigcup\{[\![\pi_2]\!]_{\mathcal{M}}(B) \mid B \in [\![\pi_1]\!]_{\mathcal{M}}(A)\}$.

5. $[\![(\pi_1 \Rightarrow \pi_2)]\!]_{\mathcal{M}}(A) =$

$$\begin{cases} \{\epsilon\} & \text{if there is a state B} \in [\![\pi_1]\!]_{\mathcal{M}}(A) \\ & \text{with} [\![\pi_2]\!]_{\mathcal{M}}(B) = \{\epsilon\} \\ \{A\} & \text{if for all } B \in [\![\pi_1]\!]_{\mathcal{M}}(A) \\ & \text{it holds that} [\![\pi_2]\!]_{\mathcal{M}}(B) \not\subseteq \{\epsilon\} \\ \emptyset & \text{otherwise.} \end{cases}$$

6. $[\![(\neg\pi]\!]_{\mathcal{M}}(A) =$

$$\begin{cases} \{\epsilon\} & \text{if } [\![\pi]\!]_{\mathcal{M}}(A) = \{\epsilon\} \\ \{A\} & \text{if } [\![\pi]\!]_{\mathcal{M}}(A) = \emptyset \\ \emptyset & \text{otherwise.} \end{cases}$$

7. $[\![\eta x : \pi]\!]_{\mathcal{M}}(A) =$

$\bigcup\{[\![\pi]\!]_{\mathcal{M}}(A[x := d]) \mid d \in \mathcal{M}\}$.

8. $[\![\iota x : \pi]\!]_{\mathcal{M}}(A) =$

$$\begin{cases} [\![\pi]\!]_{\mathcal{M}}(A[x := d]) & \text{for the unique } d \text{ with} \\ & [\![\pi]\!]_{\mathcal{M}}(A[x := d]) \not\subseteq \{\epsilon\} \\ & \text{if } d \text{ exists} \\ \{\epsilon\} & \text{otherwise.} \end{cases}$$

More information on this definition can be found in [2]. For present purposes it is sufficient to note that a DPL program can execute in three different ways, when acting on a given input state:

1. The program reports success by producing at least one proper output state. For example, the program $man(v_1)$ when acting on an input state where v_1 refers to John will succeed and return the input state as its only output state.
2. The program reports failure by not producing any output at all. For example, the program $\eta v_1 : woman(v_1)$ will fail for any input state (except ϵ) if there are no women in the model under consideration (its output state set will be empty).
3. The program reports error by producing ϵ as its only output. For example, the program $\iota v_1 : manager(v_1)$ will produce ϵ for any input state if the model under consideration does not have a unique manager.

3 Preconditions of DPL programs

Above, we have referred to DPL formulas as programs. We are now going to use tools for programming language analysis on DPL. We will use quantified dynamic logic over DPL to describe the preconditions for success, failure and error of DPL programs.

QDL terms $t ::= c \mid v$,

QDL programs $\pi ::= t = t \mid Rt \cdots t \mid (\pi;\pi) \mid (\pi \Rightarrow \pi) \mid (\neg\pi) \mid \eta v : \pi \mid \iota v : \pi$,

QDL formulas $\varphi ::= t = t \mid Rt \cdots t \mid (\varphi \wedge \varphi) \mid \neg\varphi \mid \exists v\varphi \mid \langle \boldsymbol{\pi} \rangle \varphi \mid [\boldsymbol{\pi}]\varphi$.

Note that the QDL programs are precisely the DPL programs. An atomic relation $Rt_1, \cdots, t_n$ can occur inside a QDL program or as an atomic QDL formula, so we need to distinguish the programs of QDL from the static QDL relations. We use boldface for the test program $\boldsymbol{Rt_1 \cdots t_n}$ and italics for the formula $Rt_1 \cdots t_n$.

We omit outermost parentheses as usual, and use $\top$ for a formula which is always true, $\bot$ for a formula which is always false.

The semantics of QDL is as for first order logic, with the following clauses for the program modalities added (assume $A \neq \epsilon$):

- $\mathcal{M} \models_A \langle \boldsymbol{\pi} \rangle \varphi$ iff there is some B with $B \in [\![\pi]\!]_{\mathcal{M}}(A)$, $B \neq \epsilon$ and $\mathcal{M} \models_B \varphi$.
- $\mathcal{M} \models_A [\boldsymbol{\pi}]\varphi$ iff for all $B \in [\![\pi]\!]_{\mathcal{M}}(A)$ it holds that $B \neq \epsilon$ and $\mathcal{M} \models_B \varphi$.

Note that $\langle \boldsymbol{\pi} \rangle$ and $[\boldsymbol{\pi}]$ are not duals. $\langle \boldsymbol{\pi} \rangle \top$ expresses the conditions for success of π, $[\boldsymbol{\pi}]\bot$ the conditions for failure of π. It follows that $\neg\langle \boldsymbol{\pi} \rangle \top \wedge \neg[\boldsymbol{\pi}]\bot$ expresses the conditions for error.

The following axiom schemata can be used to compute these conditions as formulas of FOL.

1. $\langle \boldsymbol{Rt_1 \cdots t_n} \rangle \varphi \leftrightarrow (Rt_1 \cdots t_n \wedge \varphi)$.
2. $[\boldsymbol{Rt_1 \cdots t_n}]\varphi \leftrightarrow (Rt_1 \cdots t_n \rightarrow \varphi)$.
3. $\langle \boldsymbol{t_1 = t_2} \rangle \varphi \leftrightarrow (t_1 = t_2 \wedge \varphi)$.
4. $[\boldsymbol{t_1 = t_2}]\varphi \leftrightarrow (t_1 = t_2 \rightarrow \varphi)$.
5. $\langle \boldsymbol{\pi_1;\pi_2} \rangle \varphi \leftrightarrow \langle \boldsymbol{\pi_1} \rangle \langle \boldsymbol{\pi_2} \rangle \varphi$.
6. $[\boldsymbol{\pi_1;\pi_2}]\varphi \leftrightarrow [\boldsymbol{\pi_1}][\boldsymbol{\pi_2}]\varphi$.
7. $\langle \neg\boldsymbol{\pi} \rangle \varphi \leftrightarrow (\varphi \wedge [\boldsymbol{\pi}]\bot)$.
8. $[\neg\boldsymbol{\pi}]\varphi \leftrightarrow (\langle \boldsymbol{\pi} \rangle \top \vee (\varphi \wedge [\boldsymbol{\pi}]\bot))$.
9. $\langle \boldsymbol{\pi_1 \Rightarrow \pi_2} \rangle \varphi \leftrightarrow (\varphi \wedge [\boldsymbol{\pi_1}]\langle \boldsymbol{\pi_2} \rangle \top)$.
10. $[\boldsymbol{\pi_1 \Rightarrow \pi_2}]\varphi \leftrightarrow (([\boldsymbol{\pi_1}](\langle \boldsymbol{\pi_2} \rangle \top \vee [\boldsymbol{\pi_2}]\bot)) \wedge ([\boldsymbol{\pi_1}]\langle \boldsymbol{\pi_2} \rangle \top \rightarrow \varphi))$.

11. $\langle \boldsymbol{\eta v} : \boldsymbol{\pi} \rangle \varphi \leftrightarrow \exists v \langle \boldsymbol{\pi} \rangle \varphi$.

12. $[\boldsymbol{\eta v} : \boldsymbol{\pi}] \varphi \leftrightarrow \forall v [\boldsymbol{\pi}] \varphi$.

13. $\langle \boldsymbol{\iota v} : \boldsymbol{\pi} \rangle \varphi \leftrightarrow (\exists! v \langle \boldsymbol{\pi} \rangle \top \wedge \exists v \langle \boldsymbol{\pi} \rangle \varphi)$.

14. $[\boldsymbol{\iota v} : \boldsymbol{\pi}] \varphi \leftrightarrow (\exists! v \langle \boldsymbol{\pi} \rangle \top \wedge \forall v (\langle \boldsymbol{\pi} \rangle \top \rightarrow [\boldsymbol{\pi}] \varphi))$.

The most interesting item of this list is the universal schema for ι assignment (item 14). To see what it means, note that $[\boldsymbol{\pi}]\top$ expresses that all output states of π are proper. The schema states that the following are equivalent:

- For proper input state A, the program $\iota v : \pi$ does only have proper output states, and all of these satisfy φ.
- For proper input state A there is precisely one $d \in M$ for which π has a proper output on input $A[v := d]$, and for all d' for which π has proper outputs on $A[v := d']$, all outputs of π on $A[v := d']$ are proper and satisfy φ.

It is not very difficult to see that these are indeed equivalent, so the axiom schema is sound, as are the other axiom schemata.

The axiom schemata can be used to calculate the truth, falsity and error conditions of DPL programs as formulas of FOL. If we represent a first order model as a database, then evaluation of DPL in a model reduces to evaluation of first order formulas in the database.

An example will make clear how the axioms may be used to compute preconditions of DPL programs as FOL formulas. Consider example (21) with translation (22).

(21) *If a woman is married,*
her husband looks after her.

(22) $(\eta x : Wx;\ Mx) \Rightarrow (\iota y : Hyx;\ Lyx)$.

Here is the derivation of the truth conditions.

$$
\begin{aligned}
& \langle (\boldsymbol{\eta x} : \boldsymbol{Wx};\ \boldsymbol{Mx}) \Rightarrow (\boldsymbol{\iota y} : \boldsymbol{Hyx};\ \boldsymbol{Lyx}) \rangle \top \\
\leftrightarrow\ & [\boldsymbol{\eta x} : \boldsymbol{Wx};\ \boldsymbol{Mx}] \langle \boldsymbol{\iota y} : \boldsymbol{Hyx};\ \boldsymbol{Lyx} \rangle \top \\
\leftrightarrow\ & [\boldsymbol{\eta x} : \boldsymbol{Wx}][\boldsymbol{Mx}] \langle \boldsymbol{\iota y} : \boldsymbol{Hyx} \rangle \langle \boldsymbol{Lyx} \rangle \top \\
\leftrightarrow\ & \forall x [\boldsymbol{Wx}][\boldsymbol{Mx}] \langle \boldsymbol{\iota y} : \boldsymbol{Hyx} \rangle \langle \boldsymbol{Lyx} \rangle \top \\
\leftrightarrow\ & \forall x (Wx \rightarrow (Mx \rightarrow \langle \boldsymbol{\iota y} : \boldsymbol{Hyx} \rangle \langle \boldsymbol{Lyx} \rangle \top)) \\
\leftrightarrow\ & \forall x (Wx \rightarrow (Mx \rightarrow \\
& (\exists! y \langle \boldsymbol{Hyx} \rangle \top \wedge \exists y \langle \boldsymbol{Hyx} \rangle \langle \boldsymbol{Lyx} \rangle \top))) \\
\leftrightarrow\ & \forall x (Wx \rightarrow (Mx \rightarrow \\
& (\exists! y Hyx \wedge \exists y (Hyx \wedge Lyx)))).
\end{aligned}
$$

To calculate the falsity conditions, we can use theorem (23), which is derivable from the axiom schemata:

(23) $[\boldsymbol{\pi}_1 \Rightarrow \boldsymbol{\pi}_2] \bot \leftrightarrow ([\boldsymbol{\pi}_1] \top \wedge \langle \boldsymbol{\pi}_1 \rangle [\boldsymbol{\pi}_2] \bot)$.

Applying theorem (23), we get the following falsity conditions for (22):

$$\exists x(Wx \wedge Mx \wedge \exists! y Hyx \wedge \forall y(Hyx \rightarrow \neg Lyx)).$$

Program (22) aborts with error if it doesn't succeed and doesn't fail. Modulo some FOL reasoning the conditions for this are given by (24):

(24) $\exists x(Wx \wedge Mx \wedge \neg\exists! y Hyx)$.

This means that in all models where married women do have unique husbands, program (22) will never abort with error. In other words, the calculus allows us to derive that the presupposition of the definite description has been cancelled by the implication.

4 The Implementation

The parser The grammar for our fragment uses categorial feature unification, and the parser is based on standard techniques for such grammars. The syntax consists of a lexicon, which associates categories with lexical items, a category descriptor which gives definitions of complex categories in terms of simpler categories and some reduction rules.

Basic categories are S without features, and E with features for number, person, case, *uindex* for up index (= antecedent index) and *dindex* for down index (= anaphor index). Complex categories are built with / and \ and the constraints on feature unification in the usual way. The index features *uindex* and *dindex* also occur on noun phrases and determiners. Here are some examples of complex categories (* marks the feature values that do not matter).

- N(number) =
  S/E(number,*,*,*).
- NP(number,person,case,uindex,dindex) =
  S/(E(number,person,case,uindex,dindex)\S).
- VP(number,person,*) =
  E(number,person,Nom,*,*)\S.
- TV(number,person,tense) =
  VP(*,*,tense)/NP(*,*,Acc,*,*).
- DET(number,uindex,dindex) =
  NP(number,Third,*,uindex,dindex)/
  N(number).
- AUX(number,person) =
  VP(number,person,Tensed)/
  VP(number,person,Inf).
- NEG =
  AUX(number,person)\AUX(number,person).

Basic categories get assigned in the lexicon. For example:

word	Category
John^i	NP(Sg,Third,*,i,*)
he$_i$	NP(Sg,Third,Nom,*,i)
him$_i$	NP(Sg,Third,Acc,*,i)
sees	TV(Sg,Third,Tensed)
a^i	DET(Sg,i,*)
the^i	DET(*,i,*)
another^i_j	DET(*,i,j)
his^i_j	DET(*,i,j)
man	N(Sg)
with	(N(number)\N(number))/NP(*,*,*,*,*)

Some examples of complex category formation:

- a man^i:
  DET(Sg,i,*) · N(Sg) =
  NP(Sg,Third,*,i,*)/N(Sg) · N(Sg) =
  NP(Sg,Third,*,i,*).
- sees a man^i:
  TV(Sg, Third, Tensed) · NP(Sg,Third,*,*,*)
  = (VP(Sg,Third,Tensed)/
  NP(Sg,Third,Acc,*,*,*))
  ·NP(Sg,Third,*,*,*)
  = VP(Sg,Third,Tensed).
- John^j sees a man^i:
  NP(Sg,Third,*,j,*) · VP(Sg,Third,Tensed) =
  (S/(E(Sg,Third,*,j,*)\S))·
  (E(Sg,Third,Nom,*,*)\S)
  = S.

The translator The translator uses λ-calculus to translate parse trees into DPL programs. We could have translated on the fly, building translations while parsing, but the present set-up seemed preferable for reasons of modularity of design.

The translation algorithm makes use of a lexical function mapping pairs consisting of a lexical item with an associated category to λ-expressions in the lexicon, along the lines of [9].

Translating a sentence into DPL boils down to lambda reduction of the lambda expression which results from combining the lambda expressions associated with the leaves of the parse tree, according to the rules of functional application dictated by the categorial structure.

Here are some examples of lambda expressions associated with lexical items with categories. Note that we assume the presence of indices in the lexicon, so we can handle anaphoric links by co-indexing.

For a proper understanding of the translation instructions one should bear in mind the distinction

between DPL variables that are used for DPL assignment and lambda calculus variables. We use lower case for the first and upper case for the latter.

Translation for *man*, N(Sg):

$$\lambda V_1.man(V_1).$$

Translation for $John^i$,NP(Sg,Third,*,i,*):

$$\lambda V_1.v_i := J; V_1(v_i).$$

Translation for *sees*, TV(Sg,Third,Tensed):

$$\lambda V_1.(\lambda V_2.(V_1 \lambda V_3.see(V_2, V_3))).$$

Translation for *is*, TV(Sg,Third,Tensed):

$$\lambda V_1.(\lambda V_2.(V_1 \lambda V_3.V_2 = V_3)).$$

Translation for $a(n)^i$, Det(Sg,i,*):

$$\lambda V_1.(\lambda V_2.(\eta v_i : V_1(v_i); V_2(v_i))).$$

Translation for the^i, Det(*,i,*):

$$\lambda V_1.(\lambda V_2.(\iota v_i : V_1(v_i); V_2(v_i))).$$

Translation for the^i_j, Det(*,i,j) (the anaphoric use of *the*):

$$\lambda V_1.(\lambda V_2.\iota v_i : (v_i = v_j; V_1 v_i); V_2(v_i)).$$

Translation for *if*, (S/S)/S:

$$\lambda V_1.(\lambda V_2.V_1 \Rightarrow V_2).$$

Translation for *does*, AUX(Sg,Third):

$$\lambda V_1.V_1.$$

Translation for *not*, NEG:

$\lambda V_1.(\lambda V_2.\neg(V_1 V_2))$.

Translation for $another^i_j$, DET(Sg,i,j):

$\lambda V_1.(\lambda V_2.(\eta v_i : v_i \neq v_j; V_1(v_i); V_2(v_i)))$.

Translation for $the\ other^i_j$, DET(*,i,j):

$\lambda V_1.(\lambda V_2.(\iota v_i : (v_i \neq v_j; V_1(v_i)); V_2(v_i)))$.

Translation for he_i, NP(Sg,Third,Nom,*,i):

$\lambda V_1.V_1(v_i)$.

Translation for his^i_j, DET(*,i,j):

$\lambda V_1.(\lambda V_2.\iota v_i : V_1(v_i); isof(v_i, v_j); V_2(v_i))$.

Translation for *with*,(N(number)\N(number))/NP(*,*,*,*,*):

$\lambda V_1.(\lambda V_2.(\lambda V_3.(V_2(V_3); V_1(\lambda V_4 isof(V_4, V_3)))))$.

All these translations are typed, but we have left most of the typing discipline implicit. For example, the translations of noun phrases all are of the type of (dynamic) generalized quantifiers, which take a property to give a DPL program. The translation of proper names is a dynamic variation of the Montague treatment for proper names [8]. In extensional Montague grammar, proper names translate into expressions denoting the set of properties which are true of the named individual. Here, proper names translate into expressions that for every property give the DPL program which first assigns the name of the individual to the index variable of the proper name, and then tests for the property. This is like in Montague grammar, but with a dynamic touch added. Anaphoric links to the name remain possible by means of the index variable as long as its value remains unchanged.

Other noun phrases with a dynamic flavour are indefinite and definite descriptions. Indefinite descriptions translate into expressions that for every property give the DPL program which does an indefinite assignment to an index variable and tests for the property. Definite descriptions are handled likewise, but with definite assignment instead of indefinite assignment.

(25) $John^1$ *uses his* pc^2_1.

As an example, we treat the translation of (25), which is obtained starting from the following parse tree:

$(S\ ,$

$\quad (NP^1, John)\ ,$

$\quad (VP,$

$\quad\quad (TV, uses),$

$\quad\quad (NP_1^2,$

$\quad\quad\quad (DET_1^2, his),$

$\quad\quad\quad (N, pc)$

$\quad\quad)$

$\quad)$

$)$

The translation step by step:

his $\text{pc}_1^2 \rightarrow \lambda V_1.(\lambda V_2.\iota v_2 :$
$V_1(v_2); \textit{is-of}(v_2, v_1); V_2(v_2))(\lambda V_1.pc(V_1))$
$\rightarrow$
$\lambda V_2.\iota v_2 : (\lambda V_1.pc(V_1))(v_2); \textit{is-of}(v_2, v_1); V_2(v_2)$
$\rightarrow$
$\lambda V_2.\iota v_2 : pc(v_2); \textit{is-of}(v_2, v_1); V_2(v_2).$

uses his $\text{pc}_1^2 \rightarrow$
$\lambda V_1.(\lambda V_2.V_1(\lambda V_3.use(V_2, V_3)))$
$(\lambda V_2.\iota v_2 : pc(v_2); \textit{is-of}(v_2, v_1); V_2(v_2))$
$\rightarrow$
$\lambda V_2.(\lambda V_2.(\iota v_2 : pc(v_2); \textit{is-of}(v_2, v_1); V_2(v_2))$
$(\lambda V_3.use(V_2, V_3))$
$\rightarrow$
$(\lambda V_2.\iota v_2 :$
$pc(v_2); \textit{is-of}(v_2, v_1); (\lambda V_3.use(V_2, V_3))(v_2))$
$\rightarrow$
$\lambda V_2.\iota v_2 : pc(v_2); \textit{is-of}(v_2, v_1); use(V_2, v_2).$

John^1 uses his $\text{pc}_1^2 \rightarrow$
$(\lambda V_1.v_1 := J; (V_1(v_1))$
$\lambda V_2.\iota v_2 : pc(v_2); \textit{is-of}(v_2, v_1); use(V_2, v_2)$
$\rightarrow$
$v_1 := J;$
$(\lambda V_2.\iota v_2 : pc(v_2); \textit{is-of}(v_2, v_1); use(V_2, v_2))(v_1)$
$\rightarrow$

$v_1 := J; \iota v_2 : pc(v_2); is\text{-}of(v_2, v_1); use(v_1, v_2).$

In the same way, (26) gets translated into (27).

(26) $John^1$ *is a* man^2.

(27) $v_1 := J; \eta v_2 : man(v_2); v_1 = v_2.$

Note that 'is' is treated as in Montague grammar [8].

5 Experiences with the Gödel Implementation Language

The declarative semantics of Gödel improves on the semantics of Prolog: extra-logical predicates (such as *var*, *nonvar*, *assert*, *retract*, !, ...) are avoided and sometimes replaced by declarative counterparts.

Like Lambda Prolog [7], Gödel is a typed language: it is necessary to declare the type and domain of all functions and predicates (polymorphism is allowed, however). This convention makes program writing a bit more cumbersome. For example, we have to declare the type Program for representing a DPL program. For each DPL statement, it is necessary to define a function to build a Program (example: Piota : Variable * Program → Program). We also have to declare a type Expression for λ-expressions. Some complications arise from the fact that an expression may contain a DPL program and vice versa. On the plus side, more errors are detected during compilation, the compiler generates more efficient code, and the typing discipline makes for more legible, comprehensible programs. Last but not least, the typing discipline has obliged us to think a bit more about the clauses we were writing than we perhaps would have done otherwise.

Gödel has facilities that permit elegant meta-programming. In Prolog the program and the meta-program are not independent: the predicates *assert* and *retract* modify the program itself in which these predicates occur. In Gödel, program and meta-program are completely independent. It is possible for a program to load another program, to modify this other program by inserting or retracting predicates, functions or types, and to demonstrate a goal. In our implementation we use these facilities to represent a model as a logic database and a precondition as a complex goal.

6 The Program Itself

The main module takes a sentence or text as input and produces a report containing the sentence, the parse tree, the DPL program it gets translated into, and the preconditions. This module uses the following submodules:

- the parser module which from a sentence, finds its category and builds its parse tree,
- the translation module which from the parse tree, computes a representation of a DPL program,

- the precondition module which from a DPL program, derives the preconditions (this module calls another module to simplify the resulting FOL formulas),
- the evaluation module which performs a database evaluation.

A lexicon module is called by the first two of these modules. It contains the words, with their categories and the associated λ-expressions.

6.1 Main module

This module receives a sentence represented by a list of words and parses it, translates it, produces a report, computes preconditions and evaluates these in a given model.

6.2 Output

This module defines how to output programs, expressions, categories, trees, words, ... It uses the facilities of Gödel for manipulating text files.

6.3 Lexicon

The lexicon is defined by a predicate *Dict* with three arguments: the word itself, a category and an appropriate lambda expression.

6.4 Parser

The parser employs backtracking and unification in the usual way. Gödel (as all logic programming languages) has these features built in, which makes it very easy to implement a parser for a simple fragment like ours. The parsing of a sentence consists of three steps:

- generate a list of categories corresponding to the sequence of words,
- reduce the list of categories,
- test if you have a sentence else retrace your steps and try again.

We use the type *categor* to represent categories. It is defined by the constant S and the functions

E(number,person,case,uindex,dindex),

NP(number,person,case,uindex,dindex),

N(number),

and so on. The two infix functions / and \ serve to build new and more complex categories.

Sentences are parsed by building a binary parse tree in bottom-up fashion. The binary parse trees are represented by a constant *Empty* and a function A. *Empty* represents the empty tree and the function A gives the information at the current node, the left subtree, and the right subtree. The information content of the nodes is of two kinds: internal nodes carry the result of combining the categories of the subtrees and leaf nodes carry a pair consisting of a word and its category.

The parse trees are built during the reduction of the list of categories, starting with a list of trees corresponding to the words of the sentence. When we reduce two adjacent categories, we replace the two corresponding trees T_1, T_2 by a single tree with T_1 and T_2 as immediate subtrees.

6.5 Translator

The translator uses two types: *Program* and *Expression*. The first represents a DPL-program, the second a complex λ-expression. We have left the rest of the typing of the lambda expressions implicit.

The definition of programs and lambda expressions is a bit cumbersome, for a λ-expression may contain a program and vice versa. This complication is reflected in the rules for substitution and reduction. For example, it is necessary to define the substitution of an expression for a variable in a program, the free occurrence of a variable in a program, etc. The rules of reduction are a straightforward rendering of the rules of β-reduction and γ-reduction in λ calculus. We do not handle α reduction, as we see no need for variable renaming.

The translation process employs the following predicates:

Trad This predicate translates a parse tree into a reduced λ-expression. Depending on the information at the current node of the parse tree, a lexical look-up of the translation takes place, or the translation is found by reducing the application of the translations of the left and right subtrees.

Trans This predicate translates a list of parse trees for the sentences of a text into the corresponding DPL program. It uses the predicate *Trad* to translate each sentence, and then links these translations by applications.

Canred This predicates takes a λ-expression and reduces it using the declarative functional semantics of λ-calculus.

6.6 From DPL to QDL

We have seen that DPL programs are represented as Gödel functions. The reduction of DPL to FOL by means of QDL gets implemented by defining reduction predicates corresponding to the QDL axiom schemata. These predicates call each other recursively.

- **FR(Rel(s, v))** is a relational atomic test. $(Rt_1 \cdots t_n)$
- **Fequal(t1, t2)** is an atomic test of equality of the terms t1 and t2. $(t_1 = t_2)$
- **Fand(phi1, phi2)** is the conjunction of two formulas. $(\varphi_1 \wedge \varphi_2)$
- **For(phi1, phi2)** is the disjunction of two formulas. $(\varphi_1 \vee \varphi_2)$
- **Fimplic(ph1, phi2)** is the implication of two formulas. $(\varphi_1 \rightarrow \varphi_2)$
- **Fall(V(i), phi)** is the expression $(\forall v_i \varphi)$
- **Fexist(V(i), phi)** is the expression $(\exists v_i \varphi)$
- **Fonlyone(V(i), phi)** is the expression $(\exists! v_i \varphi)$
- **Fnot(phi)** is the negation of the formula φ. $(\neg\varphi)$
- **Fprecexist(pi, phi)** is the expression $(\langle\pi\rangle\varphi)$
- **Fprecuniv(pi, phi)** is the expression $([\pi]\varphi)$
- **Fpar(pi, phi)** is the expression $(\neg\langle\pi\rangle\varphi \wedge \neg[\pi]\varphi)$.

In the course of applying these predicates, formulas may get generated with obvious redundancies. We have defined a formula simplifier to remove some of these. This improves the readability of the output (the formulas are output in LaTeX format) and the performance of the database lookup on the basis of the conditions.

There is the list of simplifications handled by the module *Simple*.

- $\varphi \wedge \top \leftrightarrow \varphi$
- $\varphi \wedge \bot \leftrightarrow \bot$
- $\varphi \vee \top \leftrightarrow \top$
- $\varphi \vee \bot \leftrightarrow \varphi$
- $(\varphi \leftarrow \top) \leftrightarrow \top$
- $(\top \leftarrow \varphi) \leftrightarrow \varphi$
- $(\varphi \leftarrow \bot) \leftrightarrow \neg\varphi$
- $(\bot \leftarrow \varphi) \leftrightarrow \top$
- $\neg\bot \leftrightarrow \top$
- $\neg\top \leftrightarrow \bot$
- $(\exists! v\varphi \wedge \exists \mathrm{v}\varphi) \leftrightarrow \exists! v\varphi$.

6.7 Evaluation

The intermediate language QDL allows us to translate DPL programs into formulas of FOL. These are then evaluated in a database model, i.e., a first order model which is implemented as a Gödel database (a Gödel program). There we have a so-called meta-module *Evaluation* and an object program *Logic Database*, and the meta-program manipulates the object program. We translate first order conditions into Gödel goals, and then apply the goal to the object Gödel program, using the possibilities of meta-programming offered by Gödel. In ordinary Prolog, these things could also be done, but they would look much less elegant.

Here is an example of a Gödel model (the lines preceded by % are comment lines):

```
MODULE Model1.

IMPORT Strings, Sets.

BASE Symbol.

% We use this base for every kind of term.

CONSTANT

John, Bill, Freddy, Borsalino, Myclone: Symbol.

PROPOSITION Top.

PREDICATE

Admire, Cheer, Isof, See, Use : Symbol * Symbol;

Hat, Man, Adult, King, Pc, Manager : Symbol.

% The relations defined in the model

Admire(John,Bill).

King(Bill).

Isof(Borsalino,Bill).

Hat(Borsalino).
```

See(John,Bill).

See(Bill,John).

Manager(Bill).

Man(John).

Man(Bill).

Man(Freddy).

Adult(John).

Adult(Bill).

Pc(Myclone).

Use(John,Myclone).

Top.

% Top must be defined in every model.

---

7 Conclusion

The QDL translation discussed above only handles uniqueness presuppositions of definite descriptions. The method employed is general enough, however, to handle lots of other kinds of presupposition. Lexical presuppositions, for example, are handled in the error state semantics by a slight revision of the semantic clause for atomic tests. Being a bachelor presupposes being male and adult, so the test for bachelorhood should give error if it is performed on an entity that does not satisfy the test for being a male adult.

Formally, the revision boils down to this. Let At be the set of atomic formulae of DPL. Assume a lexical presupposition function $\mathbf{lp} : At \rightarrow DPL$ mapping each atomic test predicate of the representation language to its associated lexical presupposition, conceived as a program of the representation language. For example, here are the lexical presuppositions for bachelorhood.

$$\mathbf{lp}(\mathit{bachelor}\ x) = (\mathit{male}\ x; \mathit{adult}\ x).$$

The semantic clause for atomic relations is modified to take the function **lp** into account:

$2'$. $[\![R(t_1 \cdots t_n)]\!]_{\mathcal{M}}(A) =$

$$\begin{cases} \{A\} & \text{if } [\![\mathbf{lp}(Rt_1 \cdots t_n)]\!]_{\mathcal{M}}(A) \not\subseteq \{\epsilon\} \\ & \text{and } \mathcal{M} \models_A Rt_1 \cdots t_n, \\ \emptyset & \text{if } [\![\mathbf{lp}(Rt_1 \cdots t_n)]\!]_{\mathcal{M}}(A) \not\subseteq \{\epsilon\} \\ & \text{and } \mathcal{M} \not\models_A Rt_1 \cdots t_n, \\ \{\epsilon\} & \text{if } [\![\mathbf{lp}(Rt_1 \cdots t_n)]\!]_{\mathcal{M}}(A) \subseteq \{\epsilon\}. \end{cases}$$

This modified definition gives the success and failure of the relational test modulo the fact that the lexical presupposition of the relational test holds; if the presupposition does not hold then the test results in error.

There is no need for any other changes in the rules, for the projection of lexical presupposition is taken care of by the general principles of error percolation that are already implicit in the semantic clauses. Thus, the DPL error semantics gives us that (28) presupposes that Jan is male and adult, but that (29) only presupposes that Jan is adult.

(28) *Jan is a bachelor.*

(29) *If Jan*^{1} *is male, then he*$_1$ *is a bachelor.*

The change in the semantic clause for atomic relations is reflected in the calculus by replacing the schemata for $Rt_1 \cdots t_n$ by the following versions:

1. $\langle \mathbf{Rt_1 \cdots t_n} \rangle \varphi \leftrightarrow$
   $(Rt_1 \cdots t_n \wedge \varphi \wedge \langle \mathbf{lp}(\boldsymbol{Rt}_1 \cdots \boldsymbol{t}_n) \rangle \top)$.
2. $[\mathbf{Rt_1 \cdots t_n}] \varphi \leftrightarrow$
   $(Rt_1 \cdots t_n \rightarrow \varphi) \wedge \langle \mathbf{lp}(\boldsymbol{Rt}_1 \cdots \boldsymbol{t}_n) \rangle \top)$.

In the implementation, lexical presupposition is handled by a predicate *Lp* and a modification of the reduction predicates for the relational test axiom schemata.

Right now, we are extending the fragment to deal with other kinds of presupposition failure, in particular failure of presupposition of aspectual verbs such as *start* and *stop*.

References

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

[2] J. van Eijck. The dynamics of description. *Journal of Semantics*, 10, 1993. to appear.

[3] J. van Eijck. Presupposition failure — a comedy of errors. Manuscript, CWI, Amsterdam, 1993.

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

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

[6] P.M. Hill and J.W. Lloyd. The Gödel report. Technical report, Department of Computer Science, University of Bristol, Bristol, 1991 (revised 1992).

[7] D.A. Miller. A logic programming language with lambda abstraction, function variables and simple unification. In P. Schroeder-Heister, editor, *Extensions of Logic Programming*. Springer, 1990.

[8] 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.

[9] R. Muskens. Anaphora and the logic of change. In J. van Eijck, editor, *Logics in AI / European Workshop JELIA '90 / Amsterdam, The Netherlands, September 1990 / Proceedings*, Lecture Notes in Artificial Intelligence 478, pages 412–427. Springer Verlag, 1991.

[10] R. Muskens. Tense and the logic of change. Manuscript, University of Tilburg, 1992.

[11] V. Pratt. Semantical considerations on Floyd–Hoare logic. *Proceedings 17th IEEE Symposium on Foundations of Computer Science*, pages 109–121, 1976.