JAN VAN EIJCK AND NISSIM FRANCEZ

Verb-phrase Ellipsis in Dynamic Semantics

1. INTRODUCTION

In this paper, we study Verb-phrase Ellipsis (VPE) and show its relationship to presupposition and its failure. We represent VPE by means of a dynamic semantics approach, using an extended version of Dynamic Predicate Logic (Groenendijk and Stokhof, 1991) which includes local variables and procedures. We focus on the representation of the strict/sloppy ambiguity in connection with ellipsis constructs, which comes "for free" in a procedural context. We also provide independent justification for our representation of VPE in terms of procedure definitions and their invocations. The relationship of VPE and presupposition sheds some new light on the arguments about the question whether VPE is syntactic or semantic in nature, as well as on the relative order of meaning determination and presupposition determination.

As far as we know, the relation between VPE and presupposition has not been considered before in the literature. On the other hand, the idea to look at VPE as procedure invocation is not new; it can be found, e.g., in (Gardent, 1991). Our formalization is new, though. We show that the famous ambiguity in VPE, namely the distinction between *strict* readings vs. *sloppy* readings, is related to presuppositions and their failure.

We formalize our theory using an extension of *Dynamic Predicate Logic* (DPL). The extension we propose includes the following constructs:

- Local variables
- Procedures with simple parameters (without recursion)
- Scope rules for free variables in procedures (dynamic vs. static scoping).

To make the formal connection with presupposition, we combine these extensions with a partial or 'error abortion' semantics for DPL, in the style of (Van Eijck, 1993; Van Eijck, 1994b).

We would like to stress one methodological issue, not directly related to presuppositions and their relation to VPE. Incorporating procedures into a dynamic semantics account of VPE has a two-fold advantage:

1. It provides a succinct representation of VPE which naturally accounts for the famous ambiguity of strict/sloppy readings of certain VPEs, and moreover, it does so in terms of scope-rules that are well understood in programming language semantics. The binding mechanisms involved turn out to have an independent justification in a broader context, rather than being an ad-hoc construction for VPE representation only.
2. It provides new grounds for a possible answer to a question often raised in the literature, where it tends to generate a heated argument:

   Is VPE resolution a *syntactic* process, depending on syntactic reconstruction of unrealized syntactic material, or is it a *semantic* process, *directly* interpreting the ellipsis based on some semantic representation of the realized syntactic material?

   Two recent papers, taking opposite views on this issue, are (Dalrymple, 1991) for the semantic view and (Lappin, 1993) for the syntactic view.

   A question often asked by the proponents of the syntactic view, raised as an argument against the semantic view, is the following:

   How can a semantic representation be assigned to an unrealized syntactic construct?

   In the debate, this question is a rhetorical one, of course. On the syntactic view, this can simply not be done. According to this syntactic view, it is mandatory first to reconstruct the unrealized syntactic material by a process of 'copy and paste', and then to apply ordinary semantic interpretation to the reconstructed material.

   Of course, procedures and their invocations (in programming languages) are an example *par excellence* of a construct exhibiting exactly the puzzling characteristics being questioned.

> A procedure is a *one definition/multiple use* construct. The procedure definition is assigned some meaning by the semantics of the programming language. Then, *every* single invocation is *directly interpreted*: it inherits the meaning of the procedure, possibly adapted in accordance with parametrization and scope considerations. The view that reconstruction always takes place before interpretation of a procedure, would correspond to a *macro expansion* view of procedure invocation. This view prevailed in the early days of programming languages, but has yielded long ago to the perspective of direct interpretation with its many possibilities for richer semantic contents (e.g., recursion).
> Thus, our development can be seen as a re-enforcement of the semantic view of VPE. On the other hand, we concede to the defenders of the syntactic view that there is a key difference between procedure invocation in programming and VPE in natural language. While in programming it is always clear which procedure is being invoked, an elliptical verb phrase in natural language refers to a 'procedure' that has to be reconstructed from the previous context.

In Section 2 we extend DPL with local variables. In Section 3 we add simple procedures, plus a mechanism for dynamic and static procedure calls. In Section 4 we explain the way in which VPE can be modelled by means of procedures (in DPL) and their invocations, and the way in which the strict/sloppy ambiguity manifests itself in the dynamic vs. static binding distinction. In Section 5 we consider the relationship between VPE and presupposition, and the way this relationship is captured by our representation.

This paper grew out of (Van Eijck and Francez, 1993), which was presented at the Conference on Applied Logic, *Logic at Work* in Amsterdam, 1992. A related approach, also accounting for VP ellipsis in terms of an extension of DPL, was given in (Hardt, 1994), and presented at the Amsterdam Colloquium in 1993. Despite the superficial resemblance, the accounts are quite different. Hardt's account of VP ellipsis with sloppy identity hinges on the fact that the same DPL variable is re-used to introduce a new referent. This predicts that anaphoric reference to the previous referent becomes impossible in combination with sloppy identity, which we think is an incorrect prediction.

1 Bill loves his wife. John does too, but doesn't want to admit it to him.

Here the *him* might well be anaphorically related to Bill, while the identity is sloppy. We think Hardt cannot account for this combination of facts.

2. DPL WITH LOCAL VARIABLES

Dynamic predicate logic is the result of replacing existential quantification over a variable x by random assignment to variable x, and conjunction of formulas by sequential composition. It is a variant of first order predicate logic that handles variable binding dynamically. This fact has been exploited in natural language representation, in the analysis of so called donkey pronouns and of pronouns with an antecedent in a previous sentence. Translating *A man walked in. He sat down* as $x :=?; Mx; Wx; Sx$ avoids a problem of variable binding that occurs if existential quantifiers and ordinary conjunction are used: $\exists x(Mx \wedge Wx) \wedge S\underline{x}$.

We extend the syntax of DPL with a construction for local variable declaration. Let C be a set of constants, V a set of variables, and assume $C \cap V = \emptyset$, $c \in C, v \in V$. Also assume that C contains a special constant $\flat$. We will refer to the interpretation of $\flat$ as $\flat$, and we will use this object for initialisation (see the semantic clauses for local variable declaration below).

Definition 1 *DPL with Local Variables*

terms $t ::= v \mid c$

programs $\pi ::=$

$$v :=? \mid Rt \cdots t \mid t = t \mid (\pi;\pi) \mid (\neg\pi) \mid \iota v : \pi \mid \mathtt{new}\ \ v : \pi.$$

Since sequential composition is commutative, we will omit irrelevant brackets, and write both $(\pi_1; (\pi_2; \pi_3))$ and $((\pi_1; \pi_2); \pi_3)$ as $\pi_1; \pi_2; \pi_3$. We define **skip** as $c = c$ for some arbitrary $c \in C$. Outermost brackets are not irrelevant in definite assignments and local variable declarations, for in $\iota v : \pi$ and $\mathtt{new}\ \ v : \pi$ they delimit the scope of the unicity test and of the the new allocation for the local variable respectively.

Another useful abbreviation: we use $v := t$ as shorthand for $v :=?; v = t$. [1]

Finally, $\texttt{new}\ v_1 : \texttt{new}\ v_2 : \ldots \texttt{new}\ v_n : \pi$ will be abbreviated as $\texttt{new}\ v_1, \ldots, v_n : \pi$.

For the semantics, we distinguish between *allocations* and *memory states*. Let A be a set of addresses of storage cells. Then $(A \cup \{*\})^V$ is the set of all allocations for V. We will use l, l', ..., l_1, ..., to refer to allocations. If l is an allocation and $l(v) = *$ then we say that v has not been allocated by l, or that v has not been initialized by l, or that v is an undeclared variable under l. Otherwise the allocation of v is an address in A. Let $M = (U, I)$ be a first order model of the right signature to interpret the relation symbols of a particular DPL language. We assume that $U \neq \emptyset$, so in particular we have $\flat \in U$. U^A is the set of all memory states for M. We will use s, s', ..., s_1, ..., to refer to memory states. For a systematic study of the distinction between allocation and storage in dynamic semantics of natural language we refer the reader to (Vermeulen, 1994; Vermeulen, 1991).

We use $l(v|a)$ for the allocation that differs from l at most in the fact that the allocation-value (or: the location) for v is a, and $s(a|d)$ for the state that differs from s at most in the fact that the storage-value for a is d. If s is a state and l is an allocation, then $s \circ l$ is the partial assignment of individuals in U to variables in V which is defined as follows:

$$s \circ l(v) = \begin{cases} s(l(v)) & \text{if } l(v) \neq * \\ \uparrow & \text{otherwise.} \end{cases}$$

Because the composition of an allocation and a state is a partial assignment, we have to modify the original DPL semantics to cater for the possibility that a relation is called without an appropriate

[1]Note that under this convention $x := x$ becomes equivalent to $x :=?$, not to $x = x$. This is a deviation from standard programming language conventions, but for our present purposes it does no harm, and it reduces the number of different atomic instructions.

allocation for some variable in it. In case this happens, the call aborts with error. Thus, there are three possibilities for a DPL program π to execute, given a model M, an allocation l and a memory state s (we will say that the pair (l, s) forms the *input* for π):

- program π succeeds by producing an output pair (l', s'),
- program π indicates failure for that input,
- program π aborts with error because inside π a relation gets called with an undeclared variable.

We will use $M, (l, s), (l', s') \models \pi$ for 'π succeeds in M for input (l, s) by producing output (l', s')', and $M, (l, s) \Dashv \pi$ for 'π fails in M for input (l, s)'. Error abortion of π in M for input (l, s) will now show up as: (i) for all (l', s'): $M, (l, s), (l', s') \not\models \pi$, and (ii) $M, (l, s) \not\Dashv \pi$.

Once we have the formal machinery for error abortion in place, we might as well use it for other purposes than undeclared variable detection. We now have the means to specify error abortion conditions for any DPL program, so we can deal with any presupposition for correct execution of a DPL program. In this paper, we will only deal with the presuppositions of definite assignment, by letting $\iota v : \pi$ abort with error in case the model does not contain precisely one individual for which π succeeds. See (Van Eijck, 1993) for more information on the dynamic semantics of definite descriptions.

Here are the semantic clauses for 'DPL with local variables and error abortion'. In the clause for atomic relations we assume the notion $M \models_b At$, where At is an atom, and b is a partial assignment defined for all variables occurring in At; this is nothing other than the standard Tarskian definition of satisfaction from first order logic.

1. $M, (l, s), (l', s') \models Rt_1 \cdots t_n$ iff
   - $(l, s) = (l', s')$,
   - for all $v \in \{t_1, \ldots, t_n\}$, $l(v) \neq *$, and
   - $M \models_{sol} Rt_1 \cdots t_n$.
2. $M, (l, s) \Dashv Rt_1 \cdots t_n$ iff

- for all $v \in \{t_1, \ldots, t_n\}$, $l(v) \neq *$, and
- $M \not\models_{sol} Rt_1 \cdots t_n$.

3. $M, (l, s), (l', s') \models t_1 = t_2$ iff
   - $(l, s) = (l', s')$,
   - for all $v \in \{t_1, t_2\}$, $l(v) \neq *$, and
   - $M \models_{sol} t_1 = t_2$.
4. $M, (l, s) \dashv t_1 = t_2$ iff
   - for all $v \in \{t_1, t_2\}$, $l(v) \neq *$, and
   - $M \not\models_{sol} t_1 = t_2$.
5. $M, (l, s), (l', s') \models v :=?$ iff
   - $l = l'$,
   - $l(v) \neq *$, and
   - there is some $d \in U$ with $s' = s(l(v)|d)$.
6. $M, (l, s) \dashv v :=?$ never.
7. $M, (l, s), (l', s') \models \pi_1; \pi_2$ iff there is a pair (l'', s'') with $M, (l, s), (l'', s'') \models \pi_1$ and $M, (l''s''), (l', s') \models \pi_2$.
8. $M, (l, s) \dashv \pi_1; \pi_2$ iff either $M, (l, s) \dashv \pi_1$, or
   - there is an (l', s') with $M, (l, s), (l', s') \models \pi_1$, and
   - for every (l'', s'') with $M, (l, s), (l'', s'') \models \pi_1$ it holds that $M, (l'', s'') \dashv \pi_2$.
9. $M, (l, s), (l', s') \models \neg\pi$ iff $(l, s) = (l', s')$ and $M, (l, s) \dashv \pi$.
10. $M, (l, s) \dashv \neg\pi$ iff there is a pair (l', s') with $M, (l, s), (l', s') \models \pi$.
11. $M, (l, s), (l', s') \models \iota v : \pi$ iff
    - $l(v) \neq *$,
    - there is a $d \in U$ for which $M, (l, s(l(v)|d)), (l', s') \models \pi$, and
    - there is a unique $d \in U$ for which $M, (l, s(l(v)|d)), (l'', s'') \models \pi$ for some pair (l'', s'').
12. $M, (l, s) \dashv \iota v : \pi$ never.

13. $M, (l, s), (l', s') \models \texttt{new}\ v : \pi$ iff there are an $a \in A$, $a \notin \text{rng}\,(l)$ and an l'' with $M, (l(v|a), s(a|\flat)), (l'', s') \models \pi$ and $l' = l''(z|l(z))$.
14. $M, (l, s) \dashv \texttt{new}\ v : \pi$ iff for some $a \in A$, $a \notin \text{rng}\,(l)$, $M, (l(v|a), s(a|\flat)) \dashv \pi$.

A simple induction on the structure of DPL programs establishes that the two cases 'π succeeds in M for (l, s)' and 'π fails in M for (l, s)' are mutually exclusive.

The intuition behind the semantic clause for $\iota v : \pi$ is that the program $\iota v : \pi$ succeeds if there is a unique individual in the domain for which π succeeds, and otherwise aborts with error (for it never fails).

The intuition behind the semantic clause for the declaration of a local variable v is this. First v is made to point to a new location a, which in turn is initialized to the object $\flat$. The object $\flat$ is in no way special, and in fact the initialization with $\flat$ is not essential: all we are doing is putting in a default initialization because it looks cleaner. (Compare this with a programming language where newly declared variables of type integer are initialized to 0 by default.) Upon exiting the scope of the declaration of the local variable the old location of v is restored.

Let us now give some examples to illustrate the clauses. The DPL program $\iota x : Wxy$ could be the translation of the natural language phrase *his wife*, where y is linked to the possessive pronoun. Assume a model $M = (U, I)$ with an allocation l and a state s. Assume that x, y have been declared in the global context, so that $l(x) \neq *, l(y) \neq *$. Also assume that y has an appropriate value, i.e., $s \circ l(y)$ is an appropriate referent for *his*. Then if the model does contain a unique individual satisfying the description, i.e., if $\{d \in U \mid (d, s \circ l(y)) \in I(W)\}$ is a singleton, the program succeeds and gives an output state where that individual gets stored at location $l(x)$, while the output allocation remains unchanged. If $\{d \in U \mid (d, s \circ l(y)) \in I(W)\}$ is not a singleton the program aborts with error. Next consider the following program:

$$x :=?; Px; \texttt{new}\ x : (x :=?; Qx); Rx.$$

Again assume that x is allocated in the global context. So the variable x occurs both locally and globally. Assume the program gets executed in M for input (l, s) with $l(x) \neq *$. Then if no individual in M satisfies P, the program fails. If there are Ps in M, every output (l, s') of $x :=?; Px$ will have $s'(l(x)) \in I(P)$. Next, x is made to point to a new location, so the allocation l changes to l', and if there are Qs in M, the program $x :=?; Qx$ succeeds for (l', s'). Next, the old allocation l is restored. Finally, Rx is executed, but for the old allocation to x. Thus, the overall effect is that the program succeeds if both $I(P) \cap I(R) \neq \emptyset$ and $I(Q) \neq \emptyset$, and fails otherwise. Moreover, in all outputs (l'', s'') of the program, $l'' = l$ will hold, and if (l, s'') is an output of the program, then $s''(l(x)) \in I(P) \cap I(R)$.

Here is an axiom for local variable declaration (assume y is not free in ϕ or π):

$$\langle \texttt{new } \boldsymbol{z} : \boldsymbol{\pi} \rangle \phi \leftrightarrow (y = \flat \wedge \langle \boldsymbol{\pi}[\boldsymbol{y}/\boldsymbol{z}] \rangle \phi.$$

Example:

$$\langle \texttt{new } \boldsymbol{z} : (\boldsymbol{x} :=?; \boldsymbol{x} = \boldsymbol{z}) \rangle x = \flat$$
$$\leftrightarrow (y = \flat \wedge \langle \boldsymbol{x} :=?; \boldsymbol{x} = \boldsymbol{y} \rangle x = \flat).$$

Note that the condition on y is crucial:

$$\langle \texttt{new } \boldsymbol{z} : (\boldsymbol{z} :=?; \boldsymbol{Pz}) \rangle Py \not\leftrightarrow (y = \flat \wedge \langle \boldsymbol{y} :=?; \boldsymbol{Py} \rangle Py).$$

3. DPL WITH SIMPLE PROCEDURES

We will now extend DPL with simple procedures, simple in the sense that a procedure body cannot contain a local variable declaration, a procedure call or a procedure declaration. This rules out, among other things, recursive procedure calls. Here is the syntax of DPL with local variables and simple procedures (we use Λ for the empty string):

Definition 2 *DPL with Local Variables and Simple Procedures*

terms $t ::= v \mid c$
atoms $A ::= v :=? \mid Rt \cdots t \mid t = t$
proc-bodies $B ::= A \mid (B; B) \mid (\neg B) \mid \iota v : B$
proc-names $P ::= p \mid P'$
parameter-lists $L := \Lambda \mid vL$

proc-declarations $D ::= \texttt{proc}\ \ P(L) : B\ \ \texttt{end}$
programs $\pi\ ::=\ A \mid (\pi;\pi) \mid (\neg\pi) \mid \iota v\ :\ \pi \mid \texttt{new}\ v\ :\ \pi \mid D \mid P(t \cdots t)$.

For simplicity we have assumed that procedures have names p, p'', p'', ... Procedures have parameter lists; we will treat the parameters of a procedure as local variables of that procedure. In case the parameter list of a procedure P is empty, we will abbreviate $P()$ as P.
Assume a procedure is declared as follows:

$$\texttt{proc}\ \ p(v) : x :=?; x = v\ \ \texttt{end}.$$

The procedure can be called with $p(t)$, and the effect of that call is that the instructions $\texttt{new}\ \ v : (v := t; x :=?; x = v)$ get executed. The variable v used in the declaration of p is called a *formal* parameter of the procedure, the term t used in the invocation of p is called an *actual* parameter.

A variable in the body of a procedure which is not a formal parameter of that procedure is called a *free* variable of that procedure. In the above example, x is a free variable of the procedure p.

Of course, the effect of the procedure call will depend crucially on the binding of the free variable x, namely the allocation used to map it to a location. If the binding uses the allocation at the point of procedure declaration we say that the procedure call is *static*. If it uses the allocation at the point of procedure execution we say that the call is *dynamic*. This distinction is standard in the theory of imperative programming; see e.g., (Gries, 1981, Chapter 12).

We will first concentrate on dynamic procedure calls. To give a semantics of dynamic calls of procedures, we use *contexts*, i.e., functions from *proc-names* to *proc-bodies* $\cup$ (*parameter-lists* $\times$ *proc-bodies*). A procedure declaration will have the effect that a new procedure body (with the list of formal parameters specified by the declaration, if there are any) gets stored in the context under the procedure name. When a procedure gets called with a list of actual parameters, this list gets compared to the list of formal parameters stored in the context under the procedure name.

If the lengths are the same, local space is allocated for the formal parameters, the values of the actual parameters are assigned to the formal parameters, and the procedure body stored in the context gets called. Finally, the original allocation gets restored.

The inputs and outputs of programs now are triples consisting of a context, an allocation and a memory state. If C is a context, P a procedure name, $v_1, \ldots, v_n$ a list of formal parameters, and B a procedure body, then we use $C(P|v_1, \ldots, v_n, B)$ for the context which is like C except for the possible difference that P is mapped to $v_1, \ldots, v_n, B$.

The semantic clauses are essentially the same as before, except for the fact that contexts get modified by procedure declarations and used by procedure calls. Here is the semantics for 'DPL with local variables, simple procedure declarations, dynamic procedure calls, and error abortion':

1. $M, (C, l, s), (C', l', s') \models Rt_1 \cdots t_n$ iff
   - $(C, l, s) = (C', l', s')$,
   - for all $v \in \{t_1, \ldots, t_n\}$, $l(v) \neq *$, and
   - $M \models_{sol} Rt_1 \cdots t_n$.
2. $M, (C, l, s) \Dashv Rt_1 \cdots t_n$ iff
   - for all $v \in \{t_1, \ldots, t_n\}$, $l(v) \neq *$, and
   - $M \not\models_{sol} Rt_1 \cdots t_n$.
3. $M, (C, l, s), (C', l', s') \models t_1 = t_2$ iff
   - $(C, l, s) = (C', l', s')$,
   - for all $v \in \{t_1, t_2\}$, $l(v) \neq *$, and
   - $M \models_{sol} t_1 = t_2$.
4. $M, (C, l, s) \Dashv t_1 = t_2$ iff
   - for all $v \in \{t_1, t_2\}$, $l(v) \neq *$, and
   - $M \not\models_{sol} t_1 = t_2$.
5. $M, (C, l, s), (C', l', s') \models v :=?$ iff
   - $C = C'$, $l = l'$,

- $l(v) \neq *$, and
- there is some $d \in U$ with $s' = s(l(v)|d)$.

6. $M, (C, l, s) =\!| \; v := ?$ never.
7. $M, (C, l, s), (C, l', s') \models \pi_1; \pi_2$ iff there is a triple (C'', l'', s'') with
$M, (C, l, s), (C'', l'', s'') \models \pi_1$ and $M, (C'', l''s''), (C', l', s') \models \pi_2$.
8. $M, (C, l, s) =\!| \; \pi_1; \pi_2$ iff either $M, (C, l, s) =\!| \; \pi_1$, or
   - there is a triple (C', l', s') with $M, (C, l, s), (C', l', s') \models \pi_1$, and
   - for every (C'', l'', s'') with $M, (C, l, s), (C'', l'', s'') \models \pi_1$ it holds that $M, (C'', l'', s'') =\!| \; \pi_2$.
9. $M, (C, l, s), (C', l', s') \models \neg\pi$ iff
$(C, l, s) = (C', l', s')$ and $M, (C, l, s) =\!| \; \pi$.
10. $M, (C, l, s) =\!| \; \neg\pi$ iff there is a triple (C', l', s')
with $M, (C, l, s)$ and $(C', l', s') \models \pi$.
11. $M, (C, l, s), (C', l', s') \models \iota v : \pi$ iff
   - $l(v) \neq *$,
   - there is a $d \in U$ with $M, (C, l, s(l(v)|d)), (C', l', s') \models \pi$, and
   - there is a unique $d \in U$ with
   $M, (C, l, s(l(v)|d)), (C'', l'', s'') \models \pi$
   for some triple (C'', l'', s'').
12. $M, (C, l, s) =\!| \; \iota v : \pi$ never.
13. $M, (C, l, s), (C', l', s') \models \texttt{new } v : \pi$ iff there are an $a \in A$, $a \notin \text{rng } (l)$ and an l'' with
$M, (C, l(v|a), s(a|\flat)), (C', l'', s') \models \pi$ and $l' = l''(z|l(z))$.
14. $M, (C, l, s) =\!| \; \texttt{new } v : \pi$ iff for some $a \in A$, $a \notin \text{rng } (l)$,
$M, (C, l(v|a), s(a|\flat)) =\!| \; \pi$.
15. $M, (C, l, s), (C', l', s') \models \texttt{proc } P(v_1 \cdots v_n) : B \text{ end}$ iff
   - $l = l'$, $s = s'$, and
   - $C' = C(P|v_1, \cdots v_n, B)$.
16. $M, (C, l, s) =\!| \; \texttt{proc } P(v_1, \ldots, v_n) : B \text{ end}$ never.

17. $M, (C, l, s), (C', l', s') \models P(t_1, \ldots t_n)$ iff
    - $C = C'$, $l = l'$,
    - $C(P) = (v_1, \ldots, v_n, B)$, and
    - $M, (C, l, s), (C, l, s') \models$ $\texttt{new}\ v_1, \ldots, v_n : (v_1 := t_1; \ldots; v_n := t_n; B)$.
18. $M, (C, l, s) \mathrel{=\!\!|} P(t_1, \ldots t_n)$ iff
    - $C(P) = (v_1, \ldots, v_n, B)$,
    - $M, (C, l, s) \mathrel{=\!\!|} \texttt{new}\ v_1, \ldots, v_n : (v_1 := t_1; \ldots; v_n := t_n; B)$.

For a concrete example, let us see what happens when the following procedure gets declared and executed.

$\texttt{proc}\ p : x :=?; Px\ \texttt{end}.$

Upon declaration of p, the input context C changes to $C(p|x := ?; Px)$, for the body of the procedure gets stored under the name p. Let us call the new context C'. Next, when p is called in context C', the body of p, i.e., $C'(p)$, is extracted from context and executed. In other words, $x :=?; Px$ gets executed. The allocation for x is the allocation at the point where p gets called; this is what makes the call dynamic. In case p is mistakenly called with a parameter or a list of parameters, the call aborts with error, as no matching list of parameters is found in the context under p.

Now let us look at a case where a procedure gets declared with a formal parameter.

$\texttt{proc}\ p(z) : x :=?; Rzx\ \texttt{end}.$

When this declaration is processed, the input context C changes to $C(p|(z, x :=?; Rzx))$. When $p(t)$ is called, i.e., when p gets called with one actual parameter t, the formal parameter z and the procedure body

$x :=?; Rzx$

get extracted from context. As the number of actual parameters and the number of formal parameters do indeed match, a piece of new space is allocated to z, t is assigned to z, the procedure body $x :=?; Rzx$ is executed, and the old allocation for z is restored. Let us assume that t is a variable, $t \neq z$, and t is such that

$s \circ l(t) \neq \uparrow$ (where l and s are the current input allocation and state, respectively). Then the procedure $p(t)$ succeeds if $\{d \in U \mid (d, s \circ l(t)) \in I(R)\} \neq \emptyset$. After the procedure call the original allocation for z gets restored. Note that if p is called with actual parameter z, the call succeeds of $I(R) \neq \emptyset$ (this is because $z := z$ is equivalent to $z :=?$). Again, if p is mistakenly called without parameters, or with more than one parameter, the call aborts with error, as no match is possible with what is stored in the context under p.

Of course, in an actual implementation of an imperative language, all that gets stored when a procedure is declared is a memory address of the machine code for that procedure, because the procedure gets compiled. But this is literally a matter of encoding. By assuming that the actual body of the procedure is stored, we achieve a more abstract definition of the procedure declaration mechanism.

We now want to differentiate between dynamic and static procedure calls, and we will use P^{dyn} and P^{stat} for the purpose. This is different from the actual practice of programming, where the distinction is made at the level of language design. For example, Pascal has static binding of free procedure variables, while (the original versions of) LISP and Prolog have dynamic binding of free procedure variables. For our purposes it will turn out more useful to have both possibilities available, and to make the distinction at the point of procedure call rather than at the point of procedure declaration.

The semantics of static procedure calls is a bit more involved. Basically, we want to use the variable allocation at the point of procedure declaration rather than the allocation at the point of procedure call. This means that the allocation information should be available from the context. The simplest possible solution is to store the allocation at the time of declaration as well, and this is what we will do. We assume from now on that contexts map procedure names to pairs consisting of (i) an allocation and (ii) a list of formal parameters plus a procedure body. We will use $C_0(P)$ and $C_1(P)$ to refer to the first and second components of $C(P)$. All semantic clauses remain the same, except for:

15' $M, (C, l, s), (C', l', s') \models$ `proc` $P(v_1 \cdots v_n) : B$ `end` iff

- $l = l'$, $s = s'$, and
- $C' = C(P|(l, (v_1, \cdots v_n, B)))$.

16' $M, (C, l, s) \dashv$ `proc` $P(v_1, \ldots, v_n) : B$ `end` never.

17' $M, (C, l, s), (C', l', s') \models P(t_1, \ldots, t_n)^{dyn}$ iff

- $C = C'$, $l = l'$,
- $C_1(P) = (v_1, \ldots, v_n, B)$, and
- $M, (C, l, s), (C, l, s') \models$ `new` $v_1, \ldots, v_n : (v_1 := t_1; \ldots; v_n := t_n; B)$.

18' $M, (C, l, s) \dashv P(t_1, \ldots t_n)^{dyn}$ iff

- $C_1(P) = (v_1, \ldots, v_n, B)$,
- $M, (C, l, s) \dashv$ `new` $v_1, \ldots, v_n : (v_1 := t_1; \ldots; v_n := t_n; B)$.

17" $M, (C, l, s), (C', l', s') \models P(t_1, \ldots, t_n)^{stat}$ iff

- $C = C'$, $l = l'$,
- $C_1(P) = (v_1, \ldots, v_n, B)$, and
- $M, (C, C_0(P), s), (C, C_0(P), s') \models$ `new` $v_1, \ldots, v_n : (v_1 := t_1; \ldots; v_n := t_n; B)$.

18" $M, (C, l, s) \dashv P(t_1, \ldots t_n)^{stat}$ iff

- $C_1(P) = (v_1, \ldots, v_n, B)$,
- $M, (C, C_0(P), s) \dashv$ `new` $v_1, \ldots, v_n : (v_1 := t_1; \ldots; v_n := t_n; B)$.

Because the distinction between *static* and *dynamic* is made at the point where a procedure is called instead of at the point where it is declared, static and dynamic calls of one and the same procedure are possible.

Here is again a concrete example. Assume p is declared as follows.

`proc` $p(x) : love(x, y)$ `end`.

Then $p(j)^{dyn}$ is a call to the procedure body $love(x, y)$ after local space is allocated to x and j gets assigned to x. Because the call is dynamic, reference is made to the (current) value stored at the location of y at the point of invocation of the procedure. $p(j)^{stat}$, on the other hand, is a static call to the same procedure. In this case reference is made to the (current) value stored at what was the location for y at the point where p got declared.

4. A PROCEDURAL REPRESENTATION OF VPE

We now turn to our application of the above formal apparatus to the description of VPE. One of the most basic observations regarding VPE is the ambiguity between *strict readings* and *sloppy readings*, in the presence of an anaphoric pronoun. Consider example 2.

2 John loves his wife and Bill does too.

Most of the examples of VPE consist of a conjunctive coordination, where the full conjunct (usually the first one) is the *source* clause, while the ellided conjunct (usually the second one) is the *target* clause. Under the strict reading, the interpretation of the target clause is that Bill loves John's wife, while under the sloppy reading the interpretation is that Bill loves his own wife.

As already mentioned, a question which is often raised in the literature is, what is the source of this ambiguity? This question is also related to the question whether VPE is a syntactic or a semantic phenomenon. Below, we indicate how our representation improves on the solution to these issues, as a side effect of correctly accounting for presuppositions in VPE.

In the first accounts of VPE, where VPE is viewed as mainly a syntactic phenomenon, the ambiguity was attributed to the source clause (Sag, 1976). Thus, for the example above, two different logical forms were assigned to John loves his wife, each yielding a different target clause interpretation when completing the target clause accordingly:

$$\lambda x:\ \textit{love}\ (x,\ \textit{the-wife-of}\ (j))(j)$$

$$\lambda x:\ \textit{love}\ (x,\ \textit{the-wife-of}\ (x))(j).$$

Note that in the assumed LF language, we have represented the possessive by means of a term with a function symbol. In our extended DPL language we use ι assignments, as introduced in Section 2. Note also that we represent NL verb phrases or common nouns R with corresponding symbols R in the representation language.

In more recent approaches, where the semantic nature of VPE is stressed, the ambiguity is *not* attributed to the source clause; rather, the interpretation process itself yields multiple interpretations for the target clause. A notable example is the VPE resolution via *higher-order unification* (Dalrymple *et al.*, 1991), by which two different solutions for an equation (derived from the source clause) are obtained, each inducing a different interpretation of the target clause. In both cases, there is an intimate relationship between the strict/sloppy dichotomy and the referring anaphora/bound variable anaphora dichotomy.

In the approach of this paper, the ambiguity is pushed even further. We generate a *unique* representation of the target clause in the form of a procedure in (the extended) DPL, this representation itself having multiple interpretations according to the policy of binding free (global) variables in procedure bodies. *Static binding*, whereby free variables are bound at the point of the definition of the procedure, will yield the strict reading; *dynamic binding*, whereby free variables are bound at the point of *procedure invocation*, yields the sloppy interpretation. Of course, we now have to explain why in certain cases only one of these readings is available. Our account is in line with the idea that the strict/sloppy identity should be analyzed in terms of the difference between binding in a global or local environment.[2]

In passing, we note that interpreting the procedure as even more global (to both the defining environment and to the invoking environment) generates the *deictic* interpretation of example 2, accounting for the (well-known) fact that the pronoun may refer deictically to the *same* referent in both clauses.

[2]Compare the concluding remarks of (Gardent, 1991), where this distinction is mentioned but not worked out formally.

We note here, that in the realm of programming languages, the procedure construct has two major characteristics:

- **Abstraction**: The procedure encapsulates the details of representation.
- **Single definition - multiple use**: The procedure is defined once only, but activated (possibly parametrized) as often as needed.

It is especially this second characteristic that suggests representing VPE by means of procedures and their invocations. Ellipsis can be viewed as a way of referring more than once to a construct, though in contrast to what is the case in programming languages, surface structure contains no *explicit* definition of the construct. The definition of an appropriate procedure has to be derived from the first use of the procedure, so to speak, during interpretation, and included in the semantic representation.

By having the ambiguity (when present) associated with the (unique) representation of the target clause, we model the semantic representation in a way much closer to the surface structure of the sentence, and we view this as an advantage over previous accounts. Our improvement is in line with the modern 'dynamic' approach to natural language semantics, by which semantic methods for programming languages are successfully adapted to natural language.

Suppose we want to construct a representation of Example 2 above. Without entering here into the details of a *systematic* translation from NL to DPL (see (Bouchez *et al.*, 1993; Muskens, 1991) for details) we only describe the resulting representations.

First, from the source clause the following procedure definition is obtained by abstraction:[3]

$$\texttt{proc}\ p(x) : (\iota z :\ \textit{wife-of}\,(y,\ z);\ \textit{love}\,(x,\ z)\ \texttt{end}.$$

[3] The actual mechanism of abstraction involves the source clause, the target clause(s) and parallelism among their components, very much like the higher-order unification mechanism used to solve equations derived from the clauses in (Dalrymple *et al.*, 1991). For the use of parallellism in explaining empirical facts about VPE see (Prüst, 1991).

The procedure has one formal parameter, x, representing the subject. The procedure also has one *global (i.e., free)* variable y, the binding of which depends on the binding semantics chosen. This global variable represents the pronoun in the source sentence. Finally, the procedure has a *local* (i.e., bound) variable z, for which definiteness is represented by means of the ι operator. This local variable is also the representation of the object of the clause. The procedure definition is abstracted from the following representation for the source clause.

$$y := j;\ \iota z:\ \textit{wife-of}\,(y,\ z);\ \textit{love}\ (j,\ z).$$

As the procedure p is only abstracted afterward, when a case of VPE dictates it, we can assume the following structure for the source clause plus the target clause.

$$\begin{array}{l} y := j;\ \iota z:\ \textit{wife-of}\,(y,\ z);\ \textit{love}\ (j,\ z);\\ \texttt{proc}\ p(x) : \iota z:\ \textit{wife-of}\,(y,\ z);\ \textit{love}\ (x,\ z)\ \texttt{end};\\ \texttt{new}\ y : (y := b;\ p(b)). \end{array}$$

First the first verb phrase is represented, then a procedure declaration gets extracted, then a new allocation for the global variable y of the procedure takes place, and finally the procedure is called with as actual parameter the constant representing the subject of the target clause.

The global variable y acts much as a discourse marker does in DRT (Kamp, 1981). As the procedure declaration is abstracted in retrospect, so to speak, when the presence of the elliptical construct demands it, we assume that the procedure is available to be called to interpret the VP anaphor when the representation for the target clause is constructed. This representation looks very much like that of the source clause, but now b gets assigned to y and $p(x)$ gets called with actual parameter b. If we substitute the body of p for the procedure name in this representation we see how similar the representations of the two clauses really are. This similarity of representation is also separately motivated by the need to form *cascaded VPE* (an example appears in the next section).

We claim that the strict/sloppy ambiguity of this example now can be accounted for in terms of static versus dynamic procedure call. The representation for the strict reading becomes:

$y := j;\ \iota z:\ \textit{wife-of}\,(y,\ z);\ \textit{love}\ (j,\ z);$
$\texttt{proc}\ p(x) : \iota z:\ \textit{wife-of}\,(y,\ z);\ \textit{love}\ (x,\ z)\ \texttt{end};$
$\texttt{new}\ \ y : (y := b;\ p(b)^{stat}).$

Assume that all variables are declared in the global context. We first look at the procedure invocation $p(b)^{stat}$. What happens here is that the global variable y is interpreted with respect to the allocation l at the point of procedure declaration. Assume $l(y) = a_1$ and l gets changed to l' by the local declaration of y. Suppose $l'(y) = a_2$. Then the assignment command $y := j$ has the effect of storing the referent for j in memory cell a_1, and the assignment command $y := b$ has the effect of storing the referent for b in memory cell a_2. When the procedure $p(b)$ gets called, the allocation at the point of procedure declaration is used, so y points again at memory cell a_1, and y is interpreted as *John*, which gives the strict reading.

Now assume that the procedure representing the target clause is called dynamically. The representation then becomes:

$y := j;\ \iota z:\ \textit{wife-of}\,(y,\ z);\ \textit{love}\ (j,\ z);$
$\texttt{proc}\ p(x) : \iota z:\ \textit{wife-of}\,(y,\ z);\ \textit{love}\ (x,\ z)\ \texttt{end};$
$\texttt{new}\ \ y : (y := b;\ p(b)^{dyn}).$

Now the assignment commands $y := j$ and $y := b$ have the same effect as before, of course, but when the procedure gets called, use is made of the allocation at the point of invocation, i.e., of the allocation l'. So y points at memory cell a_2, and y is interpreted as *Bill*, which gives the sloppy reading.

As mentioned before, it is also possible to get a deictic reading, in case y, the representation of the pronoun *his*, refers to an outside individual instead of to the subject of the source clause.

Note that in the simpler cases where no free pronoun is used in a VPE, our representation produces procedures without global (i.e., free) variables in their body. This predicts the unambiguity in the interpretation of such simple VPEs.

3 John loves Mary and Bill does too.

The resulting abstracted procedure now is:

$\texttt{proc}\ p(x) :\ \textit{love}\ (x,\ m)\ \texttt{end}.$

The representation of the target clause is, as before:

$$\mathtt{new}\ y : (y := b;\ p(b)).$$

Since the body of this procedure has no free occurrence of y, the static or dynamic calling of $p(b)$ makes no difference. Thus, in such cases the redundant local variable declaration can be optimized away (similar to simplification obtained in Montague semantics by applying λ-reductions).

Note that the whole representation here is oversimplified in a certain way, for the sake of not complicating the discussion by orthogonal issues. The problem is the locality of the *object* variable (to the procedure body), rendering this object unaccessible to anaphoric reference outside the procedure. In reality, such anaphoric references are possible, very similarly to subject accessibility. Thus, consider

4 John loves his wife. Bill does too. She is beautiful.

Clearly, *she* refers here anaphorically to the object of the second sentence in the sequence above. It seems that in the preferred reading of the whole sequence, the use of *she* is disambiguating, imposing a strict interpretation of the VPE. The less preferred interpretation is the sloppy reading of the VPE, where *she* refers to the object of the second sentence. We believe that the sloppy reading where *she* refers to John's wife is unavailable.

To accomodate the accessibility of the object in the procedural interpretation, a certain extension is needed. We augment the procedure with another formal parameter, of a **result** type, to which the local object is assigned, and hence accessible outside the procedure. The full representation of the abstracted procedure now has the following form:

$$\mathtt{proc}\ p(x;\ u):\ \iota z:\ \textit{wife-of}\,(y,\ z);\ \textit{love}\,(x,\ z);\ u := z\ \mathsf{end}.$$

This procedure gets called with actual parameters as $p(b;\ v)$, where the value of v is the object of the target clause. The fact that this results in a new value for v which remains available after the procedure call explains the ruling out of the anaphoric reference of *she* to John's wife in the sloppy case.

But note that this extension goes beyond the parameter mechanism that was defined in section 3. For simplicity, we stick to the original representation in the rest of the paper.

There are several situations, mentioned in the literature, in which one of the two interpretations of VPE is *blocked*, resulting in an enforced disambiguation. In the next section, we identify *presupposition failure* as one source of blocking VPE ambiguity (not considered hitherto in the literature). Most of the reasons for blocking mentioned are either syntactic, or semantic and related to licensing anaphoric reference, e.g., to quantified NPs, or to negation. We show how our procedural representation handles those situations. For example:

5 Every man loves his wife and John does too.

Here only the sloppy reading is available, due to the impossibility of an anaphoric reference to the quantified NP. The representation of the source clause for the quantified case is this:

$$\neg \mathtt{new}\ \ x : (x :=?; man\ (x); \\ \neg(y := x; \iota z : \textit{wife-of}\ (z,\ y); love\ (x, z))).$$

This representation uses dynamic negation, which, incidentally, is not a conventional imperative programming construct. The local declaration for x remains within the scopes of the negation operator, and the value of x, and hence of y, does not exist outside of the negation. Hence, any attempt to impose a strict reading would result in attempting to access the value of an uninitialized variable, and is thereby ruled out.

Another example where only a sloppy reading is available is

6 John despises himself and Bill does too.

The absence of a strict reading is due to the fact that the source clause has a reflexive pronoun instead of an ordinary one. In the interpretation process, the recognition of a reflexive pronoun results in a different procedure being the result of the abstraction. We get:

$$\mathtt{proc}\ \ p(x) :\ \ love\ (x,\ x)\ \ \mathtt{end}.$$

This procedure also has no free occurrence of a variable in its body. Thus, in the target clause representation

$$\texttt{new } y;\ (y := b;\ p(b))$$

the assignment to y has no effect, and the meaning obtained is equivalent to

$$love\ (b,\ b),$$

the expected sloppy one.

A different kind of blocking is presented in (Prüst *et al.*, 1991; Prüst, 1991), where the discourse structure, and in particular the *parallelism* induced by it, blocks one of the readings (or more, in case of the greater ambiguity arising in discourse-related VPE).

7 John likes his hat. Fred likes it too, and Susan does too.

Here only the strict reading is available for the third clause, because of the chained ellipsis, whereby the second clause (which has no sloppy reading) enforces the strictness on the third clause (which would be ambiguous if following the first clause directly). The details here are similar the the previous example where overiding of assignments took place.

A genuine limitation of our representation is its inability to handle a certain kind of *plural* pronouns. Consider

8 John loves his wife. Bill does too. They are faithful husbands.

It would be impossible to generate a reference to 'they' since nowhere are *both* subjects *coaccessible.* A more significant extension is needed to cope with this problem, left for a different opportunity.

As a final note about procedural interpretation, we would like to point out another advantage it has, independently of VPE representation. Consider the sentence

9 *Mary avoids herself.

This sentence is considered unacceptable because 'avoid' is assumed to require non-coreferring arguments (at least for a simple context like the example). This would be reflected in our representation as a restriction (derived from lexical information) on procedural abstractions derived from 'avoid'. They have to have the form

$$p(x):\ avoid\ (x,\ y)$$

with the condition $l(x) \neq l(y)$, where l is the allocation which is in force at the point of procedure invocation; this condition prohibits procedure invocations violating it. The condition and the blocking imposed by it are commonly known as the *no aliasing* condition for procedures in programming languages. It is often assumed to obtain simpler program verification. Thus, we see again a connection to a familiar phenomenon, independently justified. By the way, it is very tempting to regard this condition as a (lexical) presupposition of 'avoid' (and its procedural representation), rather than part of its meaning.

5. VPE AND PRESUPPOSITION

In this section we present an informal account of the relationship between VPE and presupposition (and its failure) as we view it. We briefly recapitulate some of the main background issues of presupposition theory, as needed for presenting our views. The basic fact about them can be introduced with the following example.

10 John loves his wife.

As is well-known, in order to assign a truth value to 10, the following *presupposition* needs to hold:

11 John has a (unique) wife.

In case 11 does not hold, 10 is said to be *undefined*, or to exhibit a truth-value gap. One way to set a grounds for a theory of presuppositions is to abandon two-valued logics in favour of three-valued logics, or partial-logics. A more recent approach (van Eijck, 1994b) uses a dynamic semantics with *error states* for that purpose, where the partiality is pushed to the dynamic level, so to speak. We view here the treatment of presupposition as a semantic problem, in contrast to other views that consider it to belong to pragmatics.

There are two basic issues studied in presupposition theories, as described below. Both of them will turn to need modifications to correctly apply to VPE.

Projection: The projection problem is the problem of the determination of the presupposition of a compound sentence, given the presuppositions of its components, as well as their meanings. Thus, a successful solution to the projection problem should be formulated in terms of a compositional presupposition theory. Note that no information (either linguistic or other) that does not arise from the components may be used in computing the projected presupposition.

Cancellation: Here the main issue is to account for *cancellation* of presupposition depending on *contextual* information. By context here we shall mean other sentences from the discourse from which the analyzed sentence is drawn.

5.1. *Presupposition Projection and VPE*

When trying to calculate the presupposition of VPE sentences like 2, one immediately faces a problem: in attempting to apply whatever rule is available for projecting conjunctions, the second conjunct (the target clause) *does not have an independent presupposition*, similarly to not having an independent meaning. Two conclusions follow.

1. The determination of presupposition can not be done solely by projection. At least for VPE, a process of presupposition *inducing* (of that of the target clause by that of the source clause) takes place. Presupposition of the target clause depends on the meaning of the source clause.
2. The processes of meaning determination and presupposition determination can not be ordered, as suggested in (Van der Sandt and Geurts, 1991) (which does not consider VPE), but have to be interleaved. As seen from the previous point, presupposition of the target clause may depend on the meaning of the source clause. However, as will be shown here, the opposite also holds: the meaning of the target clause may depend on presuppositions of the source clause.

In fact, in our formalization of our DPL extension in section 3, we implicitly have provided the means to incorporate presupposition (and its failure) in our procedural representation process. Note that a procedure like

$$\texttt{proc}\ p(x) : \iota y : \textit{wife-of}\,(y, x) \textit{love}\ (x, y)\ \texttt{end},$$

that is, the procedure to love one's wife, does have a presupposition, because the definite assignment construction

$$\iota y : \textit{wife-of}\,(y, x)$$

which is part of it has a partial semantics stating that it succeeds if a unique wife for x can be found, and aborts with error otherwise. Thus, the presupposition associated with $p(x)$ is:

$$\exists! y\ \textit{wife-of}\,(y,\ x).$$

Note that the presupposition has the same free variable as the procedure, and in fact, it follows from the error abortion semantics for DPL that the following principle holds:

> A presupposition associated with a procedure is always evaluated in the same binding as the procedure itself.

Thus, if the procedure is invoked statically after a new variable declaration and assignment, with actual parameter b, with

$$\texttt{new}\ \ y : (y := b;\ p(b)^{stat}),$$

the presupposition is that the referent of y at the point of declaration has a unique wife, and if the procedure is invoked dynamically, again after a declaration plus assignment, with actual parameter b, with

$$\texttt{new}\ \ y : (y := b;\ p(b)^{dyn}),$$

the presupposition is that the referent of y at the point of call has a unique wife.

Thus, in both cases, first the presupposition of the target clause gets determined. Only then can projection take place and can the presupposition of the whole be computed. Under the strict reading, we get just the shared presupposition of the two components. Under the sloppy reading, we get (assuming a natural conjoining rule for conjunction of the two presuppositions): both John and Bill are assumed to have unique wives.

Next, we turn to presupposition failure. Suppose that the contextual information implies that Bill is a bachelor. In this case, the presupposition under the sloppy reading would fail. We stipulate that under such circumstances, the sloppy reading is blocked, and the ambiguity resolved in favour of the strict reading. Thus, the meaning may depend on presupposition failure. A question, which at this point we leave as unresolved, is what happens in case the presupposition of the source clause fails (i.e., John is a bachelor). Under the $OTAT$ assumption[4] which is implicit in the error abortion semantics for DPL, namely the principle that errors are propagated, we would expect the interpretation of the whole sentence to abort with error. This should be the case even if the target clause is computed as the sloppy reading, and possibly having a non-failing presupposition. A forteriori, this is the expected result for the strict reading, in which both presuppositions of the clauses fail.

The interplay between meaning determination and presupposition determination exhibited by these examples casts a serious doubt on attempts at the pragmatic handling of the latter. We see that the semantic machinery is necessary for that purpose. Still, this does not preclude that pragmatic factors may be involved too.

5.2. *Presupposition Failure and Meaning Postulates*

Consider another example discussed frequently in the literature:

12 Mary corrected her mother's mistake before she did.

Usually, the discussion focuses on the strict/sloppy interpretation of the pronoun 'her', where the pronoun 'she' is assumed to anaphorically refer to 'her mother'. Why is the reference of 'she' to 'Mary' ruled out? The answer is, that the sentential adverb 'before' is assumed to be non-reflexive. In other words, in a sentence of the form $p\ before\ q$, if time t_1 is associated with p and time t_2 is associated with q, $t_1 < t_2$ is assumed. On a reading in which 'she' anaphorically refers to 'Mary', we would get a meaning equivalent to that of

13 Mary corrected her mother's mistake before Mary corrected her mother's mistake,

and assuming that the two occurrences of 'mistake' refer to the same mistake, this would lead to a self-contradictory proposition

[4] Acronym for *Once a Thief, Always a Thief.*

under the non-reflexivity assumption.

In a Montagovian set-up, this assumption would be realized via a *meaning postulate*, excluding models with reflexive interpretations of 'before'. However, it is also possible to regard this assumption as a presupposition of sentences of the above form[5]. Adopting such a point of view allows for a more uniform explanations of blocking of readings, in this case for VPE. It allows also for considering *arbitrary* models, correctly predicting the interpretation in "bad" ones as an error state, reflecting presupposition failure.

6. CONCLUSIONS

In this paper, we have inspected Verb-phrase Ellipsis and investigated its relationship to presupposition and its failure. We hope that this paper adds to the understanding of both VPE and presupposition, and that it shows the advantage of dynamic semantics over more traditional LF-based representations.

We proposed a novel representation of VPE by means of a dynamic semantics approach, using an extended version of Dynamic Predicate Logic which includes procedures, scope rules and the distinction between static and dynamic procedure calls. Thus, we show that rather strong relationships between natural languages and programming languages may be pointed out, bringing to intuitively appealing ways of semantical representation. We have focused on the representation strict/sloppy ambiguity of VPE, showing that it may be viewed as resulting from the well-known distinction between static binding and dynamic binding of free variables in procedure bodies, a phenomenon often encountered in computer science. We also have provided independent justification for this representation. The main justification is in its ability to accommodate direct interpretation of constructs with non-fully-realized syntactic material. Thereby, we believe we have strengthened the case of the semantic conception of VPE interpre-

[5]Note that in such sentences, this presupposition is *not* projected from the component sentences p and q, but it enters the picture as a lexical presupposition of **before**.

tation and resolution.

Regarding presupposition failure, we showed that it causes disambiguation of VPEs with respect to the strict/sloppy distinction. This shows that meaning determination and presupposition determination are interrelated, and indeed we have seen that our extended semantics for the DPL representation language gives an account of this interrelation.

Acknowledgements This work was initiated during a visit of the second author to CWI, sponsored by a grant from project NF 102/62-356 ('Structural and Semantic Parallels in Natural Languages and Programming Languages'), funded by the Netherlands Organisation for the advancement of research (N.W.O.).

At the Technion, work of the second author was partly supported by a grant from the Israeli ministry of science, under the project 'programming languages induced computational linguistics', a grant from the Technion vice-president for research (Formal semantics for natural language) and by the fund for the promotion of research in the Technion.

Jan van Eijck
Centre for Mathematics and Computer Science (CWI),
Amsterdam, The Netherlands
and OTS, Utrecht University
Utrecht, The Netherlands

Nissim Francez
Department of Computer Science,
The Technion, Haifa, Israel.

REFERENCES

Bouchez, O., van Eijck, J. and Istace, O.: 1993, "A strategy for dynamic interpretation: a fragment and an implementation". In: Krauwer, S., Moortgat,M. and des Tombe, L. (eds), *Sixth Conference of the European Chapter of the Association for Computational Linguistics — Proceedings of the Conference*, ACL, pp. 61–70.

Dalrymple, M.: 1991, "Against Reconstruction in Ellipsis". Technical report, Palo Alto, CA: Xerox PARC.

Dalrymple, M., Shieber, S.M. and Pereira, F.C.N.: 1991, "Ellipsis and higher-order unification", *Lingustics and Philosophy* **14**, pp. 399–452.

van Eijck, J.: 1993, "The dynamics of description", *Journal of Semantics* **10**, pp. 239–267.

van Eijck, J.: 1994a, "Axiomatizing dynamic predicate logic with quantified dynamic logic". In: J. van Eijck and Visser, A. (eds), *Logic and Information Flow*, Cambridge, MA: MIT Press, pp. 30–48.

van Eijck, J.: 1994b, "Presupposition failure — a comedy of errors". To appear in *Methods of Logic in Computer Science.*

van Eijck, J. and Francez, N.: 1993, "Procedural semantics, verb-phrase ellipsis, and presupposition". Technical Report CS-R9311, Amsterdam: CWI, February, 1993. Also in: *Proceedings APLOC conference*, Dec 1992, Amsterdam.

Gardent, C.: 1991, "Dynamic semantics and vp-ellipsis". In: J. van Eijck (ed.), *Logics in AI / European Workshop JELIA '90 / Amsterdam, The Netherlands, September 1990 / Proceedings*, Berlin: Springer Verlag. Lecture Notes in Artificial Intelligence 478, pp. 251–266.

Gries, D.: 1981, *The Science of Programming.* Berlin: Springer Verlag.

Groenendijk, J. and Stokhof, M.: 1991, "Dynamic predicate logic", *Linguistics and Philosophy* **14**, pp. 39–100.

Hardt, D.: 1994, "Sense and reference in dynamic semantics". In: Dekker, P. and Stokhof, M. (eds), *Proceedings 9th Amsterdam Colloquium*, Amsterdam: ILLC, pp. 333–348.

Kamp, H.: 1981, "A theory of truth and semantic representation". In: Groenendijk, J. *et al.* (eds), *Formal Methods in the Study of Language.* Amsterdam: Mathematisch Centrum.

Lappin, S.: 1993, "The syntactic basis of ellipsis resolution". In: Berman, S. and Hestvik, A. (eds), *Proceedings of the Stuttgart Ellipsis Workshop*, Arbeitspapiere des Sonderforschungsbereichs 340. Stuttgart: University of Stuttgart.

Muskens, R.: 1991, "Anaphora and the logic of change". In: J. van Eijck (ed.), *Logics in AI / European Workshop JELIA '90 / Amsterdam, The Netherlands, September 1990 / Proceedings*, Berlin: Springer Verlag. Lecture Notes in Artificial Intelligence 478, pp. 412–427.

Prüst, H.: 1991, "On Discourse Structuring, VP Anaphora and Gapping". PhD thesis, Amsterdam: University of Amsterdam.

Prüst, H., Scha, R. and van den Berg, M.: 1991, "A formal grammar tackling verp-phrase anaphora". Technical Report CL-91-03, Amsterdam: ILLC.

Sag, I.: 1976, "Deletion and Logical Form". PhD thesis, Cambridge, MA: MIT.

van der Sandt, R.A. and Geurts, B.: 1991, "Presupposition, anaphora, and lexical content". Technical Report 185, IWBS, IBM Germany, 1991.

Vermeulen, C.F.M.: 1994, "Explorations of the Dynamic Environment." PhD thesis, Utrecht: OTS.

Vermeulen, C.F.M.: "Merging without mystery", Technical Report 70, Utrecht: University of Utrecht, Department of Philosophy, December 1991. To appear in the *Journal of Philosophical Logic*.