Counting Variables in a Dynamic Setting

Marco Hollenberg[*]
Utrecht University[‡]

Kees Vermeulen[†]
Utrecht University[‡]
Free University Amsterdam[§]

December 1994

Abstract

We discuss the issue of finite variable fragments from a dynamic perspective. Instead of taking PRED, first order logic with equality, as our base language, we look at DPLE, a variant of predicate logic developed in the area of dynamic semantics for natural language. We present a characterisation of all the finite variable fragments of DPLE.

1 Introduction

In the recent past an interesting shift of perspective has taken place in formal approaches to the semantics of natural language. Traditionally the meaning of a natural language expression was described in terms of truth conditions: to know the meaning of an expression is to know when it is true and when it is false. But in the early eighties so called *dynamic* approaches to natural language semantics arose which chose to give formal descriptions of the meaning of natural language expressions in terms of procedures. In these approaches interpreting a natural language expression is the same as executing the procedure assigned to it in the procedural or *dynamic* semantics (cd. [6], [5]).
The motivation for dynamic semantics comes from linguistics: the explanation of some well known hard problems in the semantics of *anaphora* becomes much easier once a dynamic view towards semantics is adopted.
The shift towards a dynamic semantics also gives rise to interesting questions regarding the techniques (to be) used in natural language semantics. For one, the question arises to which extent the dynamic trend can (should) work with the static machinery traditionally used in natural language semantics. This leads to interesting attempts to re-use the static machinery of traditional logic in dynamic way. This way we may hope to find good tools for dynamic semantics, but it can also happen that we gain a new perspective on the traditional machinery.
One important example of the interaction between the dynamic trend and the logical tradition is the development of procedural semantics for predicate logic. In this paper we will start from a procedural semantics for predicate logic: we will show how to read formulas of predicate logic as programs executed on stacks. Then we will consider a slightly more liberal language for writing programs on stacks, DPLE (cf. [10], [11]). We will see that DPLE is very similar to predicate logic in many

[*] *E-mail*: `hollenb@phil.ruu.nl`, *WWW*: `http://www.phil.ruu.nl/home/marco/`
[†] *E-mail*: `keesv@phil.ruu.nl`, `vermeule@cs.vu.nl`, *WWW*: `http://www.cs.vu.nl/ vermeule/`
[‡] Department of Philosophy, Heidelberglaan 8, 3584 CS Utrecht, The Netherlands
[§] Department of Mathematics and Computer Science, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands

ways, but the extra flexibility in the syntax of DPLE will allow us to express more with fewer variables. We will make this precise by a systematic investigation of the finite variable fragments of DPLE: we show that the number of variable names required for a DPLE program does not exceed the number of variables involved in the atomic programs, i.e. the arity of the predicates that occur in the program. This will imply that we can simulate any formula of predicate logic in which no predicates of arity larger than n occur with a DPLE program that contains at most n variables.
As a result we now get a *dynamic* view on what the study of finite variable fragments is about: traditionally some connection is assumed between the number of variables required to express a certain proposition and amount of memory that is required to evaluate the proposition. But in our dynamic approach we see immediately that it is not the amount of memory that is at stake: instead the number of variables required depends on the means we have to access memory. Our results will show that the number of variables that is required to perform a certain task (express a certain proposition) does not depend so much on the task involved, but is more crucially dependent on the choice of the set of basic programs and program constructs. Thinking about this in terms of the stack semantics has given rise to one very handy choice for such a set, DPLE.
This way dynamic semantics (once again) gives rise to a refreshing new view on a familiar issue.

2 Predicate Logic as a Programming Language

In this section we show how to interpret formulas of predicate logic as programs on stacks.[1] Let's first look at some of the fundamental notions on stacks that we will need.

Definition 1 (stacks) *Let a (non-empty) set D be given.*[2]

- ▷ *A (possibly empty) sequence $s \in D^*$ is called a stack.*
  We use ε to denote the empty stack.
- ▷ *If $s = d_1 \dots d_n$ and $s' = d'_1 \dots d'_m$, then we write $s * s'$ for $d_1 \dots d_n d'_1 \dots d'_m$.*
- ▷ *If $s = d_1 \dots d_n$ $(n \geq 1)$, then we write $top(s)$ for d_n.*
  In case $s = \varepsilon$, $top(s)$ is undefined.
- ▷ *If $s = d_1 \dots d_n$, then $|s| = n$, the length of the sequence s.*
- ▷ *A stack valued assignment, f, is a mapping that assigns a stack of values to each variable: $f : VAR \rightarrow D^*$.*
  We use λ as notation for the 'empty assignment': $\lambda(x) = \varepsilon$ $(x \in VAR)$.
- ▷ *$SASS_D$ is the set of stack valued assignments.*
- ▷ *For each variable $x \in VAR$ we define the relation $[x] \subseteq SASS_D \times SASS_D$ as follows:*
  $f[x]g$ iff $f(x) * top(g(x)) = g(x)$ & $f(y) = g(y)$ whenever $y \neq x$.

Note that $[x]$ pushes a random value on the x-stack of the input assignment. The converse of this 'program' pops the latest value of the x-stack. This gives us two ways of looking at a statement $f[x]g$: read in one direction it describes a (random) push action, read in the other direction it describes a pop action.
Now we are ready to define the procedural interpretation of PRED.

[1]Here and throughout the paper we will assume that $=$ is present in the language, unless we make an explicit exception.

[2]An equality statement $a = b$ between terms a and b that are possibly undefined should be read as: both a and b *are* defined and equal.

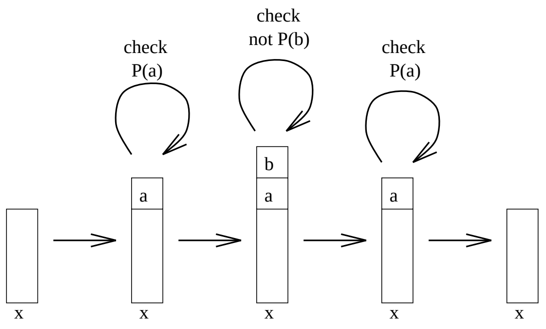

Figure 1: Left-to-right interpretation of PRED.

Definition 2 (PRED) *Let a first order model $\mathcal{M} = \langle D, I \rangle$ be given. The procedural interpretation of predicate logic assigns to each formula ϕ of predicate logic a relation $[\![\phi]\!]_{\mathcal{M}}$ on stack valued assignments in $SASS_D$, as follows (we will omit the subscript $\mathcal{M}$ if no confusion can arise):*

$$
\begin{array}{lcl}
f[\![x = y]\!]g & \Leftrightarrow & f = g \ \& \ top(f(x)) = top(f(y)) \\
f[\![P(x_1, \ldots, x_n)]\!]g & \Leftrightarrow & f = g \ \& \ (top(f(x_1)), \ldots, top(f(x_n))) \in I(P) \\
f[\![\neg\phi]\!]g & \Leftrightarrow & f = g \ \& \ \neg\exists h : \ f[\![\phi]\!]h \\
f[\![\phi \wedge \psi]\!]g & \Leftrightarrow & \exists h : \ f[\![\phi]\!]h \ \& h[\![\psi]\!]g \\
f[\![\exists x \ \phi]\!]g & \Leftrightarrow & \exists h, k : \ f[x]h \ \& \ h[\![\phi]\!]k \ \& \ g[x]k
\end{array}
$$

It can be checked easily that:

Proposition 3 *Let f be an assignment defined (at least) on the free variables of ϕ. Then:*

$$\mathcal{M}, f \models \phi \ \textit{iff} \ \exists g : \ f[\![\phi]\!]_{\mathcal{M}} g$$

(Note that this formulation makes sense: an ordinary (total or partial) assignment is just a special case of a stack valued assignment.)
We see that in this definition a *relation* is assigned to each formula of PRED. So we think of formulas $\phi \in PRED$ as nondeterministic programs. In fact the nondeterminism will never show up if we only look at these programs from the outside: whenever $f[\![\phi]\!]_{\mathcal{M}} g$ holds, we can see that $f = g$. But the nondeterminism does show up *inside* the programs, especially in the last clause where we may choose an arbitrary extension h of f such that $f[x]h$. This ensures that the $\exists$-programs already will succeed if at least one suitable x can be found.
We can now think of the evaluation of a formula of predicate logic as a run of the corresponding (nondeterministic) program on stacks. For example, in the evaluation of:

$$\exists x \ (P(x) \ \wedge \ \exists x \ \neg P(x) \ \wedge \ P(x))$$

we first put a random value for x on the x-stack. Then we check whether this x has property P. Thus the first part of the program will have a successful run iff some element $a \in I(P)$ exists. Then we go and push another value on the x-stack. For this value we check that it does *not* have the property P. Now we are at the point where we pop out of the nested quantification and we need access to the 'old' value for x. This is done by popping one value from the x-stack. Then we check (once again) whether this x has property P. Finally this value is popped off the x-stack as well. See figure 1.

Note that it is the use of stacks that enables us to evaluate the formula from left to right. An alternative would be to let $\exists x$ *replace* the value of x, but then we would get into trouble with a left to right evaluation, since we would no longer have access to the first x at the point where we have to check $P(x)$ for the second time.

3 Dynamic Predicate Logic with Exit Operators (DPLE)

In this section we present DPLE, *Dynamic Predicate Logic with Exit Operators*, and discuss its relation to PRED.[3]

In the previous section we saw how PRED can be interpreted in terms of procedures (i.e. relations) on stack valued assignments. We see that the atomic formulas correspond to tests on the values that we find on the tops of each stack. Strictly speaking these tests are the only basic programs that we encounter. But in the procedure used for $\exists$-formulas, we find two other basic operations on stacks: the push and the pop operation. In the situation of PRED these operations always occur in pairs, so that only programs of the form: $\mathsf{push}_x \ldots \mathsf{pop}_x$ occur. Furthermore, the push, pop-pairs are always neatly nested, as in:

$$\mathsf{push}_x \ldots \mathsf{push}_y \ldots \mathsf{pop}_y \ldots \mathsf{pop}_x$$

where the variable y that was the last one to be pushed, is the first one to be popped.

This restriction on the occurrence of these programs makes perfect sense, of course, if we start from predicate logic. But if we allow ourselves to think about procedures on stacks in a more independent way, then it would make at least as much sense to simply have push_x and pop_x as basic programs that we may use at any convenient time.

So let's add the push and the pop operation on stacks to the repertoire of basic procedures. The result is DPLE:

Definition 4 (DPLE)

$x = y \in DPLE$	*for any* $x, y \in VAR$
$P(x_1, \ldots, x_n) \in DPLE$	*for any* n*-ary predicate* P, $x_1, \ldots, x_n \in VAR$
$[_x \in DPLE$	*for any* $x \in VAR$
$_x] \in DPLE$	*for any* $x \in VAR$
$\neg\phi \in DPLE$	*for any* $\phi \in DPLE$
$\phi \cdot \psi \in DPLE$	*for any* $\phi, \psi \in DPLE$

Here we use $\phi \cdot \psi$ to denote the concatenation of the programs ϕ and ψ. Hence $\cdot$ in DPLE corresponds to $\wedge$ in PRED. The new atomic programs $[_x$ and $_x]$ correspond to push_x and pop_x actions respectively.

We define abbreviations for other 'boolean' connectives:

$$\bot \equiv \neg[_x \qquad \top \equiv \neg\bot \qquad (\phi \rightarrow \psi) \equiv \neg(\phi \cdot \neg\psi)$$

Now the (dynamic) semantics of DPLE can be defined as follows:

Definition 5 *Let a first order model $\mathcal{M} = (D, I)$ be given. To each formula $\phi \in DPLE$ we assign a relation $[\![\phi]\!]_{\mathcal{M}} \subseteq SASS_D \times SASS_D$ as follows:*[4]

[3]The name DPLE is chosen, because this logic was developed as an extension of DPL, as introduced in [5].

[4]We trust that this overloading of the notation $[\![\cdot]\!]$ will not cause confusion.

$$\begin{array}{lcl}
f[\![x = y]\!]g & \Leftrightarrow & f = g \ \& \ top(f(x)) = top(f(y)) \\
f[\![P(x_1,\ldots,x_n)]\!]g & \Leftrightarrow & f = g \ \& \ (top(f(x_1)),\ldots,top(f(x_n))) \in I(P) \\
f[\![[_x\ ]\!]g & \Leftrightarrow & f[x]g \\
f[\![\ _x]]\!]g & \Leftrightarrow & g[x]f \\
f[\![\neg\phi]\!]g & \Leftrightarrow & f = g \ \& \ \neg\exists h :\ f[\![\phi]\!]h \\
f[\![\phi\cdot\psi]\!]g & \Leftrightarrow & \exists h :\ f[\![\phi]\!]h \ \& h[\![\psi]\!]g
\end{array}$$

We say that a formula ϕ ∈DPLE is true in $\mathcal{M}$, notation: $\mathcal{M} \models \phi$, iff there is a $g \in SASS_D$ such that $\lambda[\![\phi]\!]g$.

Given what we have seen in the previous section, it is perfectly clear that each formula of predicate logic can also be written as a DPLE procedure: we replace $\wedge$'s by $\cdot$'s and we can simulate (existential) quantification with a combination of a $[_x$ and a $\ _x]$ program. So we immediately get a translation $^\circ$: PRED $\rightarrow$ DPLE that is truth preserving (and also respects the number of variables).
But the fact that we now have push_x and pop_x as basic programs gives us some extra flexibility in the specification of programs. In particular, we can now have:

- push_x without pop_x
- pop_x without push_x
- mixed scopes: $\mathsf{push}_x \ldots \mathsf{push}_y \ldots \mathsf{pop}_x \ldots \mathsf{pop}_y$

These three ingredients will allow us to say more with fewer variables.

4 Programming in DPLE

In this section two important programming tricks will be demonstrated that we will need later on. These programming tricks depend heavily on the use of equality and double negation, respectively.
First, consider the problem of moving the top element of the stack associated with the variable x to the top of the y-stack, where $x \neq y$. The following program does precisely this: [5]

$$\mathsf{move}_{x,y} \equiv [_y \cdot x = y \cdot \ _x]$$

Consider an assignment f with $f(x) = s * a$, $f(y) = t$. As a convention we will use s, t, u, v for sequences of elements of a particular domain D (i.e. elements of $SASS_D$) and a, b, c as elements of this domain. As we're not interested in values for variables other than x and y, we represent this assignment as the pair $(s * a, t)$. The program $[_y$ is a nondeterministic program, taking f to $(s * a, t * b)$, for *any* b. For the next instruction of the move program to succeed, however, it is imperative that b was chosen equal to a: otherwise $x = y$ would fail. So $[_y \cdot x = y$ has its unique output $(s * a, t * a)$, given input $(s * a, t)$. $\ _x]$ has the effect of removing a from the x-stack, giving us $(s, t * a)$ as required of the $\mathsf{move}_{x,y}$-program. See figure 2 for a visual illustration of the program: stacks are represented here as labelled boxes stacked on top of each other, the uppermost boxes are the final elements of the stack and the value of a specific assignment on a variable x is depicted directly above this variable.
A trickier program is $\mathsf{swap}_{x,y}$, where x is again different from y. We want this program to swap the top elements of the x- and y-stacks, and fail if either of these stacks is empty. If we allow ourselves to use another variable, z, then this is easily seen to be accomplished by the program:

$$\mathsf{swap}_{x,y} \equiv \mathsf{move}_{x,z} \cdot \mathsf{move}_{y,x} \cdot \mathsf{move}_{z,y}$$

[5] The interpreation of $\cdot$ is relational composition, which is an *associative* operation. Therefore we may, without causing confusion, write expressions such as $\phi \cdot \psi \cdot \chi$.

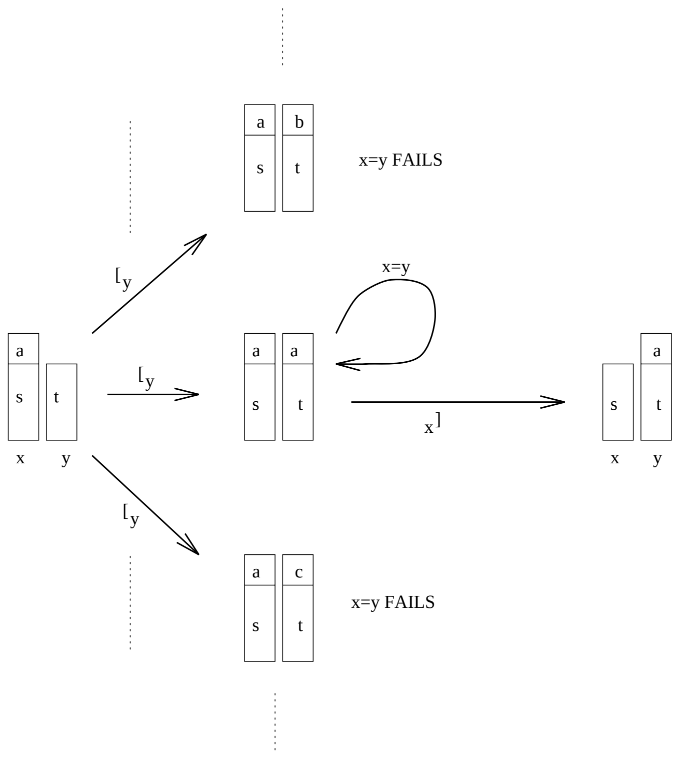

Figure 2: How $\mathsf{move}_{x,y}$ works.

For if we have an assignment represented by $(s * a, t * b, u)$, where the first, the second and the third part of the tuple are the x, y and z-stacks respectively, then:

$$\begin{array}{rcl}(s * a, t * b, u) & \overset{\mathsf{move}_{x,z}}{\longmapsto} & (s, t * b, u * a) \\ & \overset{\mathsf{move}_{y,x}}{\longmapsto} & (s * b, t, u * a) \\ & \overset{\mathsf{move}_{z,y}}{\longmapsto} & (s * b, t * a, u)\end{array}$$

What if we restrict ourselves to using just x and y in our definition? Is $\mathsf{swap}_{x,y}$ then still definable? The answer is *yes*, but not without a dirty trick: double negation.

$$\mathsf{swap}_{x,y} \equiv \mathsf{move}_{y,x} \cdot [_y \cdot \neg\neg(\ _x] \cdot x = y) \cdot \mathsf{move}_{x,y} \cdot \ _x] \cdot \mathsf{move}_{y,x}$$

This program requires some explaining. Suppose we are in a state $(s * a, t * b)$, where $s * a$ is the x-stack and $t * b$ the y-stack. Applying the program $\mathsf{move}_{y,x}$ to this input gives us $(s * a * b, t)$ as output. $[_y$ then gives us $(s * a * b, t * c)$ for any c. If $a \neq c$ then $_x]$ would give us $(s * a, t * c)$ at which point $x = y$ would fail. Thus $\neg(\ _x] \cdot x = y)$ would succeed at $(s * a * b, t * c)$, hence $\neg\neg(\ _x] \cdot x = y)$ would fail there. But if $a = c$ then $_x] \cdot x = y$ succeeds at $(s * a * b, t * c)$, so $\neg\neg(\ _x] \cdot x = y)$ also succeeds, *without* affecting the input state. So using $_x]$ within a double negation allows us to throw away things, without really losing them: the double negation makes sure the input state is unaltered. In a sense, double negation is like thinking ahead: we can wonder what would happen if we threw away the top element of the x-stack, without actually doing it!
To return to our program, we have seen that $\mathsf{move}_{y,x} \cdot [_y \cdot \neg\neg(\ _x] \cdot x = y)$ returns $(s * a * b, t * a)$ on input $(s * a, t * b)$. Now:

$$(s * a * b, t * a) \overset{\mathsf{move}_{x,y}}{\longmapsto} (s * a, t * a * b)$$

$$\begin{aligned} &\xmapsto{{}_x]} && (s, t * a * b) \\ &\xmapsto{\mathsf{move}_{y,x}} && (s * b, t * a) \end{aligned}$$

which is the desired result.
To finish off this section, we give a convention and another useful program which we will use in the next section.

Definition 6 (Iteration) *If ϕ is any DPLE-formula and $n \in \mathbb{N}$ then ϕ^n is inductively defined as follows:*

$$\phi^0 \equiv \top \qquad \phi^{n+1} \equiv \phi^n \cdot \phi$$

Another useful program is $\mathsf{copy}(x, y, n)$, where x and y are two different variables and n is a natural number. What it does is the following: on input $(s * a * t, u)$ (a pairing of an x-stack and y-stack), where $|t| = n$, it gives as output $(s * a * t, u * a)$. Should the x-stack contain less than $n + 1$ elements, the program fails. We give two versions of the program: one using an extra variable, z, but no negation, and one using just x and y but using double negation. The two programs are both denoted as $\mathsf{copy}(x, y, n)$. This will not cause any confusion: the two programs are extensionally equal: they have the same input-output pairs. We consider programs without double negation to be more in the spirit of DPLE (i.e. more resource-sensitive), so if we are allowed to use more than two variables, always assume that $\mathsf{copy}(x, y, n)$ is the version without double negation.

Definition 7 (Copy-function 1) *The copy-function without double negation, but with an extra variable z is:*

$$\mathsf{copy}(x, y, n) \equiv [_y \cdot \mathsf{move}^n_{x,z} \cdot x = y \cdot \mathsf{move}^n_{z,x}$$

On input $(s * a * t, u, v)$ (a pairing of the x-stack, the y-stack and the z-stack, where $|t| = n$), this program first puts an arbitrary element b on the y-stack, then puts t in storage on the z-stack in order to check that $a = b$ and then returns t to the x-stack.

Definition 8 (Copy-function 2) *The copy-function in DPLE($\{x, y\}$) is:*

$$\mathsf{copy}(x, y, n) \equiv [_y \cdot \neg\neg((\ {}_x])^n \cdot x = y)$$

On the same input as above, this program puts an arbitrary element b on the y-stack and checks that $a = b$ by checking that $x = y$ is able to succeed if we first remove t from the x-stack. Because of of the double negation trick it is possible to do so without actually discarding t.

5 The Finite Variable Fragments of DPLE

In this section we show, by means of examples, that in DPLE we can say more with fewer variables. Then we present a detailed characterisation of the finite variable fragments of DPLE.

Let's first convince ourselves that the extra flexibility that we have introduced in DPLE pays of: we will show, by means of examples, how we can express more with fewer variables in DPLE. The first example we consider is relation composition.[6] The definition of relation composition is a well known case where PRED requires

[6]These examples were discovered in cooperation with Albert Visser.

at least three variables (cf. [9], [7]). This means that, if we want to write down a formula with binary predicate variables R, S and U that is exactly true in models where $R = S \circ U = \{(d, d') | \exists d'' \; dSd'' \; \wedge \; d''Ud'\}$, then this formula will contain at least three variables. For example, the following formula would do:

$$\forall x \forall y \, (xRy \;\leftrightarrow\; \exists z \, (xSz \;\wedge\; zUy))$$

In DPLE we can express this sentence with only two variables: we can give a program with two variables that has a successful run in a model exactly if $R = S \circ U$. For example, we can use the formula:

$$\neg([_x \, [_y \, \neg(xRy \rightarrow \; ([_y \cdot xSy \cdot [_x \cdot x = y \cdot \;{}_y] \cdot xUy \cdot \;{}_x])) \;{}_y] \;{}_x]) \;\cdot$$
$$\neg([_x \, [_y \, \neg(([_y \cdot xSy \cdot [_x \cdot x = y \cdot \;{}_y] \cdot xUy \cdot \;{}_x] \;\rightarrow\; xRy)) \;{}_y] \;{}_x])$$

Here we see in the body of the formula an example of the mixing of scopes. Combined with the identity statement $x = y$ this allows us to move a value that was stored on the y-stack to the x-stack as soon as we no longer need to know the previous value of x. This is how we manage to avoid using a third variable z in the DPLE-definition of $R = S \circ U$.
Note that the example does not only use the fact that push's and pop's are freely available: it also uses equality statements. We will see below that it is very handy to have equality statements around if we want to work with a limited number of variables: equality statements allow us to transfer a value from one stack to another. Therefore they are very important in finite variable fragments.
But there are also examples where they are not crucial to get an improved expressivity. Consider for example:

$$\exists x \exists y \exists u \exists v \, (x < y \;\wedge\; y < u \;\wedge\; u < v \;\wedge\; v < x)$$

This sentence says that the interpretation of $<$ contains a 'square': $\Box(xyuv)$. In PRED we can express this in three variables (check!), but it is well known that we cannot express it with two variables. In DPLE however we have the following formula which does exactly that:

$$[_x \cdot [_y \cdot x < y \cdot [_x \cdot y < x \cdot \; [_y \cdot x < y \cdot \;{}_x] \cdot \; y < x \cdot \;{}_x] \cdot \;{}_y] \cdot \;{}_y]$$

Here we obtain the extra expressivity purely by mixing the scopes.
So we see that in these cases DPLE requires fewer variables than PRED to express propositions over first order models. At this point we start to wonder how much exactly can be said with a limited number of variables in DPLE. We will see below that DPLE reaches an optimum here: if we start from basic programs that contain at most n distinct variables, then any complex program of DPLE can be simulated by a program with only n variables, that is successful in exactly the same start states. Below we will give a more precise statement of this fact and we will also provide a detailed proof. But before we embark on that mission we can already make a few preliminary observations.
First it is not to be expected that we will be able to express very much with less than n variables: since we have basic programs that use n variables, then this involves a test on n stacks simultaneously. It is unlikely that, in the general case, less than n stacks will allow for a sensible simulation of this (basic) program. (This will be made precise below.) The only way to improve on this would be by fiddling with the basic programs, which is not allowed in DPLE.
Secondly, now that we have the picture of the evaluation of formulas on stacks before our mind's eye, it is not really surprising that n variables suffice: from the examples we have seen that the mixing of scopes, in combination with identity, allows us to move about information freely from one stack to another. So we can

imagine that, whenever we have a basic program $P(x_1, \ldots, x_n)$ to execute, we can simply start shuffling the values on the stacks until the relevant information is on the top. Then we can evaluate the basic program and re-shuffle all the stacks back into their original position. This is in fact the technique that we will use below and we will see how the technique only depends on the presence of a few basic shuffle programs in the DPLE language.
Below we will be working in a lot of different fragments of DPLE, so it will be convenient to have some fixed notation for all these fragments. Given any language $\mathcal{L}$ we will use the notation $\mathcal{L}^n$ to denote the restriction (reduct) of $\mathcal{L}$ to the formulas that do not contain predicate letters P^k of arity $k > n$. For each (finite) set $X \subseteq VAR$ we will write $\mathcal{L}(X)$ for those formulas of $\mathcal{L}$ that only contain the variables in X (free or bound). Again, also $\mathcal{L}(X)$ will be a language in the usual sense. In case we are only interested in the cardinality of X we will write $\mathcal{L}_{|X|}$.
So we will write, for example, DPLE$^n(X)$ for the formulas of DPLE that only use variables in X and that contain no predicates of arity larger than n.[7]

5.1 Reducing the number of variables

Suppose we are in DPLE^n, a DPLE-language with equality and with predicates of arity at most n. As equality is a binary predicate, this n will be ≥ 2. In this section we will prove that in such a language any formula using *more* than n variables can be simulated using just n variables. To be precise: consider a set of variables $X = \{x_1, \ldots, x_m\}$, where $m > n$. Choose another set of (distinct) variables $Y = \{y_1, \ldots, y_n\}$. We can then define a translation-function:

$$\langle \_ , \_ \rangle^\diamond : \mathbb{N}^m \times \text{DPLE}^n(X) \rightarrow \mathbb{N}^m \times \text{DPLE}^n(Y)$$

with the following properties.
Given a model $\mathcal{M}$ and an assignment f let $f^\diamond$ be the assignment defined as follows:

$$f^\diamond(z) := \begin{cases} f(x_1) * \ldots * f(x_m) & \text{if } z = y_1 \\ \varepsilon & \text{else} \end{cases}$$

Let $\mathrm{R}_f = (|f(x_1)|, \ldots, |f(x_m)|) \in \mathbb{N}^m$. This R_f keeps track of how the values for x_i in f are stored in $f^\diamond(y_1)$. Given a formula $\phi \in \text{DPLE}^n(X)$ we can apply the translation-function, giving us a pair $\langle \mathrm{R}_f, \phi \rangle^\diamond = \langle \mathrm{R}', \phi' \rangle$. Now:

1. $f[\![\phi]\!]g$ in $\mathcal{M}$ implies $f^\diamond[\![\phi']\!]g^\diamond$ in $\mathcal{M}$.
2. $f^\diamond[\![\phi']\!]h$ in $\mathcal{M}$ implies that there is a g such that $f[\![\phi]\!]g$ and $g^\diamond = h$.

Let us see what this means for *truth* in a model. Recall that a DPLE-formula ϕ is true in a model $\mathcal{M}$ (notation: $\mathcal{M} \models \phi$) if there is an assignment f such that $\lambda[\![\phi]\!]f$, i.e. if there is a run of the program $[\![\phi]\!]$ on the empty assignment. Consider a formula ϕ in DPLE$^n(X)$. Then $(\mathrm{R}_\lambda, \phi)^\diamond = ((0, \ldots, 0), \phi)^\diamond = (\mathrm{R}', \phi')$, for some $\mathrm{R}' \in \mathbb{N}^m$ and $\phi' \in \mathcal{L}(Y)$. The properties of $(\_ , \_)^\diamond$ mentioned above tell us, given a model, if $\lambda[\![\phi]\!]f$ then $\lambda^\diamond = \lambda[\![\phi']\!]f^\diamond$ and if $\lambda^\diamond = \lambda[\![\phi']\!]g$ then we can find an f with $\lambda[\![\phi]\!]f$. So, as any formula uses only a finite number of variables:

Proposition 9 $\forall \phi \in \mathit{DPLE}^n . \exists \phi' \in \mathit{DPLE}^n(Y) . \forall \mathcal{M} . \, (\mathcal{M} \models \phi \Leftrightarrow \mathcal{M} \models \phi')$

The next few subsections will be devoted to proving the prerequisites for this proposition. We will slow down somewhat and first try to give the reader some feeling for programming in DPLE. After this we will give an intuitive explanation of how the simulation works. Finally, the function $\langle \_ , \_ \rangle^\diamond$ will be given.

[7]Note that $DPLE^n$ again is a *language*, in the sense that it is closed under all the connectives that occur in the formulas of $DPLE^n$. So the notation DPLE$^n(X)$ indeed makes sense.

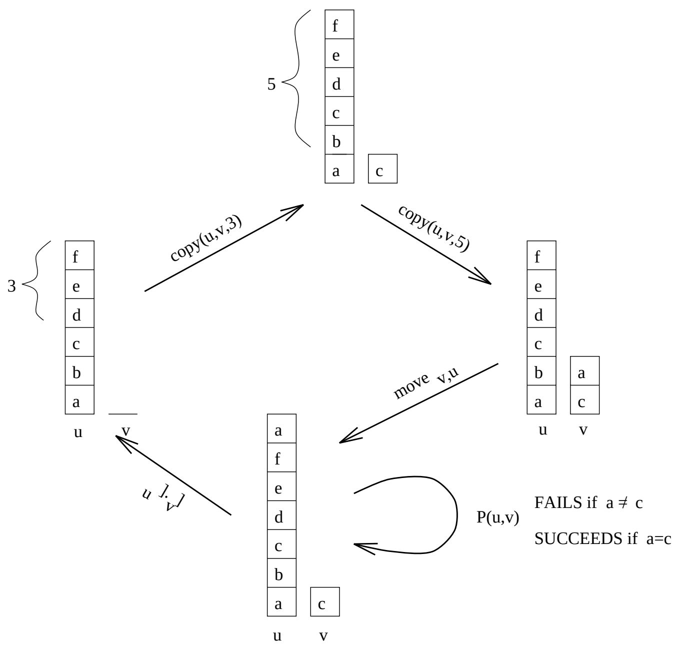

Figure 3: Simulating $P(x_1, x_2)$ in the variables u and v.

5.2 Sample simulations

Suppose we want to simulate the program $P(x_1, x_2) \cdot P(x_2, x_3)$ as it operates on the input $(a, b * c, d * e * f)$, where the x_i-stack is represented as the i-th component of this tuple, in a model $\mathcal{M}$. We are looking for a program in DPLE($\{u, v\}$) that succeeds at $(a * b * c * d * e * f, \varepsilon)$ iff the original succeeds at $(a, b * c, d * e * f)$. Furthermore, the construction of this program should only be dependent on the height of the x_i-stacks, not on their content, nor on $\mathcal{M}$. So the tuple $(1, 2, 3)$ should be sufficient information for our translation. The program we are searching for is:

$$\begin{aligned}
&\mathsf{copy}(u, v, 3) \cdot \mathsf{copy}(u, v, 5) \cdot \mathsf{move}_{v,u} \cdot \\
&P(u, v) \cdot \ _u] \cdot \ _v] \cdot \\
&\mathsf{copy}(u, v, 0) \cdot \mathsf{copy}(u, v, 3) \cdot \mathsf{move}_{v,u} \cdot \\
&P(u, v) \cdot \ _u] \cdot \ _v]
\end{aligned}$$

The first line of this program places a and c (the top elements of the x_1- and x_2-stacks in the original assignment) on the u- and v-stacks respectively. To put a on the u-stack we first place it on the v-stack and then move it to the u-stack. After this we check whether $P(a, c)$ holds in the model. If this is so we discard of all the top elements to get back to our original situation. This whole process is visualised in figure 3. A similar program simulates $P(x_2, x_3)$. Clearly this program can only succeed on $(a * b * c * d * e * f, \varepsilon)$ if $P(a, c)$ and $P(c, f)$ are true in our model. But these are the same conditions under which $P(x_1, x_2) \cdot P(x_2, x_3)$ succeeds on input $(a, b * c, d * e * f)$! So this simulation satisfies the conditions.
For a different case: what if we want to simulate $[_{x_1}$ on the same assignments, in a similar fashion? This is accomplished by the following program:

$$\mathsf{move}^5_{u,v} \cdot [_u \cdot \mathsf{move}^5_{v,u}$$

The following sequence describes the working of the program on input $(a * b * c * d * e * f, \varepsilon)$:

$$\begin{array}{rcl}(a*b*c*d*e*f,\varepsilon) & \overset{\mathsf{move}^5_{u,v}}{\longmapsto} & (a, f*e*d*c*b) \\ & \overset{[_u}{\longmapsto} & (a*g, f*e*d*c*b) \\ & \overset{\mathsf{move}^5_{v,u}}{\longmapsto} & (a*g*b*c*d*e*f,\varepsilon)\end{array}$$

where g is an arbitrary element. Notice that our translation should also keep track of how the original x_i-stacks are represented in the latter assignment. For the original assignment this information could be given by the tuple $(1,2,3)$, telling us that the x_1 stack has length 1, the x_2-stack has length 2 while that of x_3 has length 3. Using the tuple $(1,2,3)$ we are able to retrieve, or *decompose*, the original valuation from $(a*b*c*d*e*f,\varepsilon)$, at least as far as the x_1, x_2, x_3- values are concerned. If we apply $[_{x_1}$ to the original assignment $(a, b*c, d*e*f)$ the x_1-stack grows larger by one. This information should be present in the translation: we will need to change the tuple from $(1,2,3)$ to $(2,2,3)$. So our translation-function is not from formulas to formulas in less variables, but from pairs of m-tuples encoding information about the representation and formulas to pairs of m-tuples and formulas in less variables.

5.3 The translation

This subsection will define $\langle \_\,,\_\rangle^\diamond$ and prove that it has the desired properties. Throughout the translation we let R be any m-tuple of natural numbers $(\mathrm{R}_1,\ldots,\mathrm{R}_m)$: call such an R a *register*. On input of a register R and a DPLE$^n(X)$-formula ϕ our translation gives us a pair $\langle \mathrm{R}', \phi'\rangle$ of another register and a DPLE$^n(Y)$-formula. We let $\langle \mathrm{R}, \phi\rangle^\diamond_i$ be the i-th projection of the latter pair.
We will define $\langle \_\,,\_\rangle^\diamond$ and simultaneously prove the following statements, for any model $\mathcal{M}$ and any formula $\phi \in$ DPLE$^n(X)$:

1. If $f[\![\phi]\!]g$ then $f^\diamond[\![\langle \mathrm{R}_f, \phi\rangle^\diamond_2]\!]g^\diamond$ and $\mathrm{R}_g = \langle \mathrm{R}_f, \phi\rangle^\diamond_1$.
2. If $f^\diamond[\![\langle \mathrm{R}_f, \phi\rangle^\diamond_2]\!]h$ then h will send every variable but y_1 to ε. Furthermore $\langle \mathrm{R}_f, \phi\rangle^\diamond_1 = \mathrm{R} = (\mathrm{R}_1,\ldots,\mathrm{R}_m)$ will tell us how to decompose $h(y_1)$ to get an output for ϕ on input f. $h(y_1)$ will be a sequence $s_1*\ldots*s_m$, where $|s_i| = \mathrm{R}_i$. Now we can define $h_\diamond$ by
$$h_\diamond(z) := \begin{cases} s_i & \text{if } z = x_i \in X \\ f(z) & \text{else}\end{cases}$$
Clearly $(h_\diamond)^\diamond = h$. What's more, $f[\![\phi]\!]h_\diamond$.

We use the sequence $(s_1,\ldots,s_m)$ to denote an assignment f in relation to the language DPLE$^n(X)$: s_i then denotes $f(x_i)$, the x_i-stack of f. When an assignment is relevant just for the language DPLE$^n(Y)$ we denote it similarly as an n-tuple of sequences. $\vec{\varepsilon}$ denotes the appropriate number of ε's: what is appropriate will be clear from context.
Finally, we are ready to give the translation:

PUSH
$$\langle (\ldots,\mathrm{R}_i,\ldots), [_{x_i}\,\rangle^\diamond = \langle (\ldots,\mathrm{R}_i+1,\ldots), \mathsf{move}^d_{y_1,y_2}\cdot [_{y_1}\cdot \mathsf{move}^d_{y_2,y_1}\rangle$$
where $d = \sum_{j=i+1}^m \mathrm{R}_j$.

If $f[\![[_{x_i}\,]\!]g$ then the part of g relevant to DPLE$^n(X)$-formulas can be represented as $(s_1,\ldots,s_{i-1}, s_i*a,\ldots,s_{i+1},\ldots,s_m)$ for some a in the model-domain. Let $d =$

$\sum_{j=i+1}^{m}(\mathrm{R}_f)_j$. Then $d = |s_{i+1}| + \ldots + |s_m|$, so:

$$\begin{array}{lll} f^\diamond & [\![\mathsf{move}^d_{y_1,y_2}]\!] & (s_1 * \ldots * s_i, s_m^\smile * \ldots * s_{i+1}^\smile, \vec{\varepsilon}) \\ & [\![[_{y_1}\ ]\!] & (s_1 * \ldots * s_i * a, s_m^\smile * \ldots * s_{i+1}^R, \vec{\varepsilon}) \\ & [\![\mathsf{move}^d_{y_2,y_1}]\!] & (s_1 * \ldots * s_i * a * s_{i+1} * \ldots * s_m, \vec{\varepsilon}) \end{array}$$

where $s^\smile$ is the sequence s reversed. Clearly $g^\diamond = (s_1 * \ldots * s_i * a * s_{i+1} * \ldots s_n, \vec{\varepsilon})$. So $f^\diamond[\![\langle \mathrm{R}_f, [_{x_i}\ \rangle_2^\diamond]\!]g^\diamond$. Furthermore $\mathrm{R}_g = (|s_1|, \ldots, |s_{i-1}|, |s_i| + 1, |s_{i+1}|, \ldots, |s_m|) = \langle \mathrm{R}_f, \phi\rangle_1^\diamond$.
Now suppose $f^\diamond[\![\langle \mathrm{R}_f, [_{x_i}\ \rangle_2^\diamond]\!]h$. Then by the same reasoning as above: $h = (s_1 * \ldots * s_i * a * s_{i+1} * \ldots s_m, \vec{\varepsilon})$ for some a. Using $\langle \mathrm{R}_f, \phi\rangle_1^\diamond$ to decompose this assignment, we get $h_\diamond = (s_1, \ldots, s_{i-1}, s_i * a, s_{i+1}, \ldots, s_m)$ (it agrees with f on all variables not in X). Clearly $f[\![[_{x_i}\ ]\!]h_\diamond$, so we are done.

POP

$$\langle (\ldots, \mathrm{R}_i, \ldots),\ {}_{x_i}]\rangle^\diamond = \begin{cases} \langle (\ldots, \mathrm{R}_i - 1, \ldots), \mathsf{move}^d_{y_1,y_2} \cdot\ {}_{y_0}] \cdot \mathsf{move}^d_{y_2,y_1}\rangle & \text{if } \mathrm{R}_i > 0 \\ \langle (\ldots, \mathrm{R}_i, \ldots), \bot\rangle & \text{else} \end{cases}$$

where $d = \sum_{j=i+1}^{m} \mathrm{R}_j$.

Suppose $f[\![\ {}_{x_i}]]\!]g$. Then $s_i = t * a$ for some sequence t and element a, and $g = (s_1, \ldots, s_{i-1}, t, s_{i+1}, \ldots, s_m)$. Now if $d = |s_{i+1}| + \ldots + |s_m|$ then

$$\begin{array}{lll} f^\diamond & [\![\mathsf{move}^d_{y_1,y_2}]\!] & (s_1 * \ldots * s_{i-1} * t * a, s_m^\smile * \ldots * s_{i+1}^\smile, \vec{\varepsilon}) \\ & [\![\ {}_{y_1}]]\!] & (s_1 * \ldots * s_{i-1} * t, s_m^\smile * \ldots * s_{i+1}^\smile, \vec{\varepsilon}) \\ & [\![\mathsf{move}^d_{y_2,y_1}]\!] & g^\diamond \end{array}$$

Clearly $\mathrm{R}_g = \langle \mathrm{R}_f,\ {}_{x_i}]\rangle_1^\diamond$.
If $f^\diamond[\![\langle \mathrm{R}_f,\ {}_{x_i}]\rangle_2^\diamond]\!]h$ then $\langle \mathrm{R}_f,\ {}_{x_i}]\rangle_2^\diamond \neq \bot$, hence $s_i \neq \varepsilon$, say $s_i = t * a$. $\langle \mathrm{R}_f,\ {}_{x_i}]\rangle_2^\diamond$ is a deterministic program with on input $f^\diamond$ as only possible output

$$(s_1 * \ldots * s_{i-1} * t * s_{i+1} * \ldots * s_m, \vec{\varepsilon})$$

This must therefore be h. Decomposing with

$$\langle \mathrm{R}_f,\ {}_{x_i}]\rangle_1^\diamond = (|s_1|, \ldots, |s_{i-1}|, |s_i| - 1, |s_{i+1}|, \ldots, |s_m|)$$

gives us $h_\diamond = (s_1, \ldots, s_{i-1}, t, s_{i+1}, \ldots, s_m)$. So $f[\![\ {}_{x_i}]]\!]h_\diamond$.

TEST

$$\langle \mathrm{R}, P(x_{i_1}, \ldots, x_{i_k})\rangle^\diamond = \left\langle \mathrm{R},\ \begin{array}{l} \mathsf{copy}(y_1, y_k, d(i_k)) \cdot \ldots \cdot \mathsf{copy}(y_1, y_2, d(i_2)) \cdot \\ \mathsf{copy}(y_1, y_2, d(i_1)) \cdot \mathsf{move}_{y_2,y_1} \cdot \\ P(y_1, \ldots, y_k) \cdot\ {}_{y_1}] \cdot \ldots \cdot\ {}_{y_k}] \end{array} \right\rangle$$

if $\mathrm{R}_{i_j} > 0$ for all $1 \leq j \leq k$ and where $d(p) = \sum_{j=p+1}^{m}$ for each $p \in \{i_1, \ldots, i_n\}$.

$$\langle \mathrm{R}, P(x_{i_1}, \ldots, x_{i_k})\rangle^\diamond = \langle \mathrm{R}, \bot\rangle$$

if $\mathrm{R}_{i_j} = 0$ for some j with $1 \leq j \leq k$.

First of all, note that as we have assumed that we do not have predicates of arity greater than n, k must be $\leq n$. Thus the translation-step here gives us a formula in the right language, $\mathrm{DPLE}^n(Y)$, as only the variables $y_1, \ldots, y_k \in Y$ are used. Note furthermore that this case also covers the case of equality, as this is just another test. Now suppose $f[\![P(x_{i_1}, \ldots, x_{i_k})]\!]g$. Then there must be sequences t_j and elements

a_j such that $s_{i_j} = t_j * a_j$, for every $1 \leq j \leq k$, and $P(a_1, \ldots, a_k)$ holds in the model. Furthermore $f = g$, so $\langle \mathrm{R}_f, P(x_{i_1}, \ldots, x_{i_k})\rangle_1^\diamond = \mathrm{R}_f = \mathrm{R}_g$ as desired. If we let $d(p) = |s_{p+1}| + \ldots + |s_m|$ for each $p \leq m$, then:

$$\begin{array}{lcl}
f^\diamond & \llbracket \mathsf{copy}(y_1, y_k, d(i_k)) \cdot \ldots \cdot \mathsf{copy}(y_1, y_2, d(i_2)) \rrbracket & (s_1 * \ldots * s_m, a_2, \ldots, a_k, \vec{\varepsilon}) \\
 & \llbracket \mathsf{copy}(y_1, y_2, d(i_1)) \cdot \mathsf{move}_{y_2, y_1} \rrbracket & (s_1 * \ldots * s_m * a_1, a_2, \ldots, a_k, \vec{\varepsilon}) \\
 & \llbracket P(y_1, \ldots, y_k) \rrbracket & (s_1 * \ldots * s_m * a_1, a_2, \ldots, a_k, \vec{\varepsilon}) \\
 & \llbracket\ {}_{y_1}] \cdot \ldots \cdot\ {}_{y_k}] \rrbracket & f^\diamond
\end{array}$$

If $f^\diamond \llbracket \langle \mathrm{R}_f, P(x_{i_1}, \ldots, x_{i_k})\rangle_2^\diamond \rrbracket h$ then $\langle \mathrm{R}_f, P(x_{i_1}, \ldots, x_{i_k})\rangle_2^\diamond \neq \bot$, hence $(\mathrm{R}_f)_{i_j} > 0$ for all $1 \leq j \leq k$, say $s_{i_j} = t_j * a_j$. Using the reasoning of the previous paragraph, it is easy to see that $\langle \mathrm{R}_f, P(x_{i_1}, \ldots, x_{i_k})\rangle_2^\diamond$ can only succeed on $f^\diamond$ if $P(a_1, \ldots, a_k)$ is true in the model. But then $f \llbracket P(x_{i_1}, \ldots, x_{i_k}) \rrbracket f$. Furthermore h must be equal to $f^\diamond$, so decomposing h with R_f yields f itself. Thus $f \llbracket P(x_{i_1}, \ldots, x_{i_k}) \rrbracket h_\diamond$.

NEGATION

$$\langle \mathrm{R}, \neg\phi \rangle^\diamond = \langle \mathrm{R}, \neg(\langle \mathrm{R}, \phi \rangle_2^\diamond) \rangle$$

This is the first inductive case. Suppose $f \llbracket \neg\phi \rrbracket f$ while $f^\diamond \llbracket \neg(\langle \mathrm{R}, \phi\rangle_2^\diamond) \rrbracket f^\diamond$ is *not* true. Then for some assignment h: $f^\diamond \llbracket \langle \mathrm{R}, \phi \rangle_2^\diamond \rrbracket h$. By the induction hypothesis, $f \llbracket \phi \rrbracket h_\diamond$, in contradiction with the assumption that $\neg\phi$ succeeds on f. So $f \llbracket \neg\phi \rrbracket f$ implies $f^\diamond \llbracket \neg(\langle \mathrm{R}, \phi \rangle_2^\diamond) \rrbracket f^\diamond$.
For the other clause, suppose $f^\diamond \llbracket \neg(\langle \mathrm{R}, \phi \rangle_2^\diamond) \rrbracket f^\diamond$, while $f \llbracket \phi \rrbracket g$ for some g (i.e. $f \llbracket \neg\phi \rrbracket f$ is false). Then $f^\diamond \llbracket \langle \mathrm{R}, \phi \rangle_2^\diamond \rrbracket g^\diamond$, by induction. But this is in contradiction with our assumption, so $f \llbracket \neg\phi \rrbracket f$ *is* true.

COMPOSITION

$$\langle \mathrm{R}, \phi \cdot \psi \rangle^\diamond = \langle \langle \langle \mathrm{R}, \phi \rangle_1^\diamond, \psi \rangle_1^\diamond\ ,\ \langle \mathrm{R}, \phi \rangle_2^\diamond \cdot \langle \langle \mathrm{R}, \phi \rangle_1^\diamond, \psi \rangle_2^\diamond \rangle$$

This translation-step is admittedly not very easy to read, but it is in fact quite natural: if $\langle \mathrm{R}, \phi \rangle^\diamond = \langle \mathrm{R}', \phi' \rangle$ and $\langle \mathrm{R}', \psi \rangle^\diamond = \langle \mathrm{R}'', \psi' \rangle$ then $\langle \mathrm{R}, \phi \cdot \psi \rangle^\diamond = \langle \mathrm{R}'', \phi' \cdot \psi' \rangle$. So the translation embodies the idea that the output of the ϕ-step is used as an input for the ψ-step.
Now if $f \llbracket \phi \cdot \psi \rrbracket g$ then for some h: $f \llbracket \phi \rrbracket h \llbracket \psi \rrbracket g$. For reference, let $\langle \mathrm{R}_f, \phi \rangle^\diamond = \langle \mathrm{R}', \phi' \rangle$ and $\langle \mathrm{R}', \psi \rangle^\diamond = \langle \mathrm{R}'', \psi' \rangle$. By induction: $f^\diamond \llbracket \phi' \rrbracket h^\diamond$ and $h^\diamond \llbracket \langle \mathrm{R}_h, \psi \rangle_2^\diamond \rrbracket g^\diamond$. Also by induction, $\mathrm{R}_h = \mathrm{R}'$, so $h^\diamond \llbracket \psi' \rrbracket g^\diamond$. Hence $f^\diamond \llbracket \phi' \cdot \psi' \rrbracket g^\diamond$, precisely as desired.
The second desideratum of the first clause is also satisfied, for $\mathrm{R}_g = \langle \mathrm{R}_h, \psi \rangle_1^\diamond$ by the induction hypothesis, which is equal to R'', because $\mathrm{R}_h = \mathrm{R}'$ and $\langle \mathrm{R}', \psi \rangle_1^\diamond = \mathrm{R}''$.
For the verification of the second clause, let R', R'', etc. be defined as above. Suppose $f^\diamond \llbracket \phi' \cdot \psi' \rrbracket h$. Then $f^\diamond \llbracket \phi' \rrbracket g \llbracket \psi' \rrbracket h$ for some g. By the induction hypothesis, $f \llbracket \phi \rrbracket g_\diamond$, where g is decomposed with R' such that $(g_\diamond)^\diamond = g$ and $\mathrm{R}_{g_\diamond} = \mathrm{R}'$. We can now apply the induction hypothesis again, obtaining: $g_\diamond \llbracket \psi \rrbracket h_\diamond$ where the latter is the decomposition of h with the aid of R''. Thus $f \llbracket \phi \cdot \psi \rrbracket h_\diamond$.

This concludes the inductive definition of $\langle \_, \_ \rangle^\diamond$ and the simultaneous proof of the conditions stated at the beginning of this subsection. Proposition 9 follows as a result. ■

To conclude this section, let us show that proposition 9 cannot be strengthened: we cannot restrict ourselves to less than n variables if we have predicates of arity n.

Proposition 10 *Suppose we have an n-ary predicate P in our language, where $n \geq 2$ but P is not the equality. Then there is no formula in DPLE$(\{x_1, \ldots, x_{n-1}\})$ truth-equivalent to* $[_{x_1} \cdot \ldots \cdot [_{x_n} \cdot P(x_1, \ldots, x_n)$.

Proof.
We define the following two models $\mathcal{M}_1$ and $\mathcal{M}_2$ for this language. Both have as its universe the set $\{1,\ldots,n\}$. $\mathcal{M}_1$ interprets P as $P_1 = \emptyset$, $\mathcal{M}_2$ interprets it as $P_2 = \{\langle 1,\ldots,n\rangle\}$. Let $\mathcal{M}_1$ and $\mathcal{M}_2$ agree on all other predicates. By induction on formulas $\phi \in \text{DPLE}(\{x_1,\ldots,x_{n-1}\})$ we can now prove that $f[\![\phi]\!]g$ in $\mathcal{M}_1$ iff $f[\![\phi]\!]g$ in $\mathcal{M}_2$.
The only case we have to verify is the case where $\phi = P(x_{i_1},\ldots,x_{i_n})$. Because ϕ may only contain $n-1$ different variables, two of the x_{i_j} must be identical, say $x_{i_k} = x_{i_l}$. If $f[\![\phi]\!]f$ in $\mathcal{M}_1$ then each $f(x_{i_j})$ would have to be of the form $t_j * a_j$ such that $\langle a_1,\ldots,a_n\rangle \in P_1$. But this set is empty, so $f[\![\phi]\!]f$ cannot be the case. If $f[\![\phi]\!]f$ in $\mathcal{M}_2$ we should be able to write the $f(x_{i_j})$ as $t_j * a_j$ again, with $\langle a_1,\ldots,a_n\rangle \in P_2$. As $x_{i_k} = x_{i_l}$, a_k must be equal to a_l, hence $\langle a_1,\ldots,a_n\rangle \neq \langle 1,\ldots,n\rangle$ and thus $\langle a_1,\ldots,a_n\rangle \notin P_2$: contradiction. So $P(x_{i_1},\ldots,x_{i_n})$ never succeeds, in both models.
Now we can prove that there is no formula in $\text{DPLE}(\{x_1,\ldots,x_{n-1}\})$ truth-equivalent to $\phi = [_{x_1} \cdot \ldots \cdot [_{x_n} \cdot P(x_1,\ldots,x_n)$. For if there were such a formula, it could not distinguish between $\mathcal{M}_1$ and $\mathcal{M}_2$. As ϕ is true in precisely those models with a nonempty interpretation of P we have a contradiction! ■

If our only binary predicate is equality and there are no predicates of greater arity in our language, our result can also not be strengthened: a single variable does not suffice. This is a consequence of the remarks of the next section: DPLE with monadic predicates and just a single variable is exactly as expressive as predicate logic with those same predicates but without equality. In such a language no truth-equivalent can ever be found of $[_x\ [_y\ .\neg(x = y)$, which states that there are at least two different elements in the domain.

5.4 The one variable case

The remaining case is the fragment of DPLE in which only one variable name is used. This case is a bit special, since with only one variable no interesting identity statements can be made: we only get $x = x$ which is semantically equivalent to $\neg\neg(\ _x])$. As a result the one variable fragment of DPLE with identity is the same as the one variable fragment of DPLE without identity and it only makes sense to compare the expressivity of DPLE(x) with DPLE^1 without identity.
Of course this is not a special feature of DPLE, but something which already holds in PRED: if we consider PRED$_1$, then we might as well assume that we do not have identities, since in PRED all the identities we can write down with only one variable are trivially true. In fact we find that in PRED the following interesting result holds:

Proposition 11 *MPRED, monadic predicate logic (without identity) only requires one variable: for each sentence (!) $\phi \in$ MPRED there is a sentence $\phi \in$ MPRED$_1$ that has the same truth value in all models.*

Proof: We only have to consider sentences which are in prenex normal form:

$$\phi \ \equiv\ Q_n x_n \ldots Q_0 x_0\ \alpha$$

where α is in conjunctive normal form:

$$\alpha \ \equiv\ \bigwedge\{\Delta_j |\ 0 \leq j \leq m\}$$

where each Δ_j is a disjunction of literals. We may assume that all the x_i are distinct and, since we only allow monadic predicates, we can also assume that each Δ_j is written as

$$\Delta_j \ \equiv\ \bigvee\{B_{i,j} : 0 \leq i \leq n\}$$

where each $B_{i,j}$ is the disjunction of the literals that (only) contain the variable x_i. Now we note that:

Lemma 12 *Let $k \leq m$ and let ∇ only contain $x_0, \ldots, x_{n-1}$ freely. Then*

$$Q_{n-1}x_{n-1} \ldots Q_0 x_0 \ (\bigwedge\{\Delta_j |\ 0 \leq j \leq k\} \ \wedge \ \nabla)$$

is equivalent to

$$\begin{array}{l} (B_{n,k} \ \rightarrow \ Q_{n-1}x_{n-1} \ldots Q_0 x_0 \ \bigwedge(\{\Delta_j |\ 0 \leq j \leq k-1\} \ \wedge \ \nabla)) \\ \wedge \\ (\neg(B_{n,k}) \ \rightarrow \ Q_{n-1}x_{n-1} \ldots Q_0 x_0 \ (\bigwedge\{\Delta_j |\ 0 \leq j \leq k-1\} \ \wedge \ \nabla')) \end{array}$$

where

$$\nabla' \equiv \ \nabla \ \wedge \ \bigvee\{B_{k,j} |\ 0 \leq j \leq n-1\}$$

(and hence again only contains the variables $x_0, \ldots, x_{n-1}$ freely.) ■

We can use this fact repeatedly to pull all occurrences of x_n outside the scope of $Q_{n-1}x_{n-1} \ldots Q_0 x_0$ in the formula ϕ above. In the resulting formula we find that the variables $x_0, \ldots, x_{n-1}$ only occur in subsentences of the form $Q_{n-1}x_{n-1} \ldots Q_0 x_0 \ \alpha'$ again. So we may assume that these sentence can already be written with only x_0. Now we can also replace all occurrences of x_n by x_0, thus reducing the number of variables to one. ■

This shows that in sentences of monadic predicate logic the number of variables used is irrelevant: we can always rewrite 'the same sentence' with just one variable. So this also implies that these sentences can be expressed in DPLE with just one variable: we can simply translate the MPRED formula with just one variable into a DPLE sentence with just one variable in the naive way.

Perhaps it is useful to note that it is also possible to reduce the number of variables *within* DPLE[1]. The method of simulation of programs that we have discussed above does not apply straightforwardly, but it can be adapted to the one variable case if we use a slightly more complex notion of register.
Recall that our simulation technique relies on the information from a register R: the register tells us which part of $f^\diamond(y_1)$ is reserved for x_1, x_2 etc. As long as we have more than one variable we can indicate this with just a sequence of natural numbers, since throughout the procedure we can always keep $f^\diamond(y_1)$ in the format:

$$f^\diamond(y_1) = f(x_1) * \ldots * f(x_m)$$

where for each i all the x_i values are adjacent.
But if we only have one variable stack at our disposal, then we can no longer temporarily move_{y_1,y_2} away material from the stack. So we cannot add new values somewhere in the middle of the stack. Therefore we will always have to put the new value for x_i on the top of the stack and remember that this value is an x_i-value. We can do this, for example, by using as a register a sequence of (finite, disjunct) sets of natural numbers:

$$\mathrm{R} = (V_1, \ldots, V_n)$$

where each set V_i tells us which locations in the stack are reserved for the variable x_i. The highest value in V_i gives the 'current' value of x_i. If we enrich our notion of register in this way, the simulation technique can be adapted successfully.[8]

[8]There is one more inconvenience, that has to do with the simulation of $_{x_i}]$ actions: since we cannot move_{y_1,y_2} material temporarily, we cannot really remove some element from the middle of the stack. But we can remove its 'address' from the register set V_i instead. It is easy to check that this has the required effect.

6 Conclusions and further work

Now we have completed our investigation of the finite variable fragments of DPLE. We have seen that in DPLE the expressivity of the finite variable fragments is 'as good as it gets'. If we want to use atomic formulas with n-variables, then we cannot expect to get away with less then n-variables in complex formulas. But in DPLE we are also in a situation where we will never need *more* than n-variables: everything that can be said in DPLE^n can be simulated in DPLE$^n(\{y_1, \ldots, y_n\})$. If we compare this situation with what we find in PRED (cf. [4] (ch. 9-13), [1], [9], [7] and references therein), then we see that the situation is very different there. In PRED a restriction on the number of variables has serious consequences on the expressive power.
Now the translation $^\circ$:PRED$\rightarrow$DPLE enables us to 'express' all formulas of PRED^n in DPLE^n and hence to express them with just n variables after all, via this detour. This shows that what we can say in DPLE^n_n includes everything we can say in PRED^n. It is not hard to check that the converse holds as well: all we can express in DPLE^n_n, can be expressed in PRED^n.
We already pointed out above that the situation in DPLE is optimal, *provided* we do not allow fiddling with the atomic propositions. The traditional work on versions of predicate logic without variables (cf. [8], [3], [2]) can be understood as doing precisely that. From the 'stack perspective' we would describe their approach(es) as follows:

1. put all relevant values on **one** stack
2. evaluate an n-ary predicate P always on the top n elements of the stack
3. add (propositional) operators that re-arrange (permute, move, copy etc.) the elements of the stack

Because of 2 the use of variable names becomes unnecessary. We can just use the predicate letter P, since we will always mean $P(x_1, \ldots, x_n)$: P will be true precisely if the top n elements of the stack are in $I(P)$.[9] An example of an operator as intended in 3 is an operator $\rho_{i,j}$ which temporarily replaces the i^{th} value of the stack by the j^{th} value. So, for a binary relation symbol R, the proposition $\rho_{2,1}R$ will be true iff $R(x_1, x_1)$ is true iff the top of the stack is in the $I(R)$ relation to itself. To obtain full expressivity with such an approach quite a lot of operators will have to be added.
So the tradition of variable free versions of predicate logic can be understood in terms of stack semantics. But the use of stacks there is very different from the use of stacks as we have proposed it. In the variable free versions of predicate logic, the goal is to re-analyse (eliminate) the use of variable names in predicate logic and using a stack makes this possible. In the dynamic approach the stacks pop up in the dynamic analysis of the quantifiers: we interpret the $\exists x$ quantifier as an instruction to push (and later pop) a value on the x-stack.

Over the past few years a lot of research has been done already into dynamic ways of doing logic. If nothing else, this research has certainly shown that a lot of interesting work in dynamic logic still lies ahead. Here we will not try to give a broad overview of all the topics that dynamic logicians will want to investigate in the future. Instead we just point out two questions that are directly related to the present paper.

[9] Thus these approaches avoid the use of variable names altogether, but from our perspective it makes more sense to count the number of variables by counting the number of *stacks* used in the semantics. So semantically speaking these logics use **one** variable after all. It is clear that, if there is only one stack that we want to talk about, we do not need a name for it: having 0 variables (syntactically) is the same as having 1 variable (semantically).

First we notice that our results have been obtained by using a simulation technique which is defined in terms of $\mathsf{move}_{x,y}$ and $\mathsf{copy}(x, y, n)$ programs. We have used both the =-statements of DPLE and the program negation to define these important procedures.[10] It would be interesting to find out what happens if we remove (one of) these connectives from our language: what happens to the finite variable fragments in DPLE without = and/or program negation?
From the examples above we already know that some of the finite variable fragments of DPLE without negation will still be a bit more expressive than the corresponding fragments of PRED, but we suspect that in absence of identity statements (or rather: in absence of $\mathsf{move}_{x,y}$ programs) limiting the number of variables will amount to a serious restriction of the expressive power. Therefore the question arises how we can indicate the bounds of the expressivity of DPLE^n in absence of =. Here we hope to gain something from a suitable genralisation of Ehrenfeucht games.
Another topic which can be linked directly to the issue of finite variable fragments is the problem of giving a deduction system for DPLE. Apart from the question: 'How many variables do I need to state proposition ϕ?', it also makes sense to wonder: 'How many variables do I need to *prove* proposition ϕ?' We know that in predicate logic there are theorems containing n variables, the proof of which requires more than n variables (in any known deduction system). We may wonder whether we can avoid that sort of situation in DPLE. This question is wide open, since no deduction system for DPLE has been formulated yet. One obvious way of doing deductions for DPLE would be via the translation with PRED. But from the point of view of trying to work with a limited supply of variables this is probably not a good idea. So one challenge is to find a deduction system for DPLE that preserves its pleasant properties regarding limitations on the number of variables.

We can conclude that we have shown that by only a slight variation of PRED we can obtain a completely different (better?) situation regarding the finite variable fragments. This can be taken to imply several things. One option is to say that PRED, with all its peculiarities, simply **is** the logic we want to use to talk about first order models. In that case the results of this paper merely show, from a dynamic perspective, where the origins of some peculiarities of PRED lie: we can now understand the restricted expressivity of the finite variable fragments of PRED dynamically, as a consequence of the specific rules for program construction that we find in PRED. In this case we can hope that the new way of looking at the finite variable problems may suggest new ways of establishing (expressivity) results for PRED itself.
But we could also see results such as the one presented here as an indication that PRED is not the most suitable 'first order logic' in all situations. Instead dynamic approaches to logic, such as DPLE, can suggest other ways of doing first order logic that might be more suitable in specific situations. This line of thinking encourages us to keep working on (other) dynamic ways of doing logic.

Acknowledgements
We would like to thank Albert Visser and Johan van Benthem for their encouragement and support.

[10]Recall that we only really needed $\neg$ for the two variable case.

References

[1] H. Andréka, I. Németi, and J. van Benthem. Universal formulas and preservations under substructures I. unpublished manuscript, july 1993.

[2] D. Ben-Shalom. Natural language, generalized quantifiers and modal logic. In P. Dekker and M. Stokhof, editors, *Proceedings of the 9th Amsterdam Colloquium*, December 1994.

[3] M Cresswell. *Entities and Indices.* SLAP. Kluwer, Dordecht, 1990.

[4] D Gabbay et al. *Mathematical Logic and Computational Aspects*, volume 1, Temporal Logic. 1994.

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

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

[7] R. Maddux. A sequent calculus for relation algebra. *Annals of Pure and Applied Logic*, 25:73–101, 1983.

[8] W.V.O. Quine. Predicate functor logic. In J-E Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 309–315, Amsterdam, 1971. North-Holland.

[9] A. Tarski. On the calculus of relations. *Journal of Symbolic Logic*, 6:73–89, 1941.

[10] C.F.M. Vermeulen. Sequence semantics for dynamic predicate logic. *Journal of Logic, Language and Information*, 2:217–254, 1993. after: idem, Logic Group Preprint Series, 60, Department of Philosophy, Utrecht University, January 1991.

[11] C.F.M. Vermeulen. *Explorations of the Dynamic Environment.* PhD thesis, Utrecht University, september 1994.