Tableau Reasoning and Programming with Dynamic First Order Logic

Jan van Eijck, *CWI and ILLC, Amsterdam, jve@cwi.nl*

2,Juan Heguiabehere *ILLC, Amsterdam, juanh@wins.uva.nl*

3,Breanndán Ó Nualláin *ILLC, Amsterdam, bon@wins.uva.nl*

Abstract

Dynamic First Order Logic (DFOL) results from interpreting quantification over a variable v as change of valuation over the v position, conjunction as sequential composition, disjunction as non-deterministic choice, and negation as (negated) test for continuation. We present a tableau style calculus for DFOL with explicit (simultaneous) binding, prove its soundness and completeness, and point out its relevance for programming with DFOL, for automated program analysis including loop invariant detection, and for semantics of natural language. Next, we extend this to an infinitary calculus for DFOL with iteration and connect up with other work in dynamic logic.

Keywords: Dynamic Logic, First Order Logic, Assertion Calculus, Tableau Reasoning

1 Introduction

The language we use and analyze in this paper consists of formulas that can be used both for programming and for making assertions about programs. The only difference between a program and an assertion is that an assertion is a program with its further computational effect blocked off. In the notation we will introduce below: if ϕ is a program, then $(\!(\phi)\!)$ is the assertion that the program ϕ can be executed. Execution of ϕ will in general lead to a set of computed answer bindings, execution of $(\!(\phi)\!)$ to a yes/no answer indicating success or failure of ϕ.

Since the formulas of our language, DFOL, can be used for description and computation alike, our calculus is both an execution mechanism for DFOL and a tool for theorem proving in DFOL. One of the benefits of mixing calculation and assertion is that the calculus can be put to use to automatically derive assertions about programs for purposes of verification. And since DFOL has its roots in Natural Language processing (just as Prolog does), we also see a future for our tool-set in a computational semantics of natural language.

We start our enterprise by developing a theory of binding for DFOL that we then put to use in a calculus for DFOL with explicit binding. The explicit bindings represent the intermediate results of calculation that get carried along in the computation process. We illustrate with examples from standard first order reasoning, natural language processing, imperative programming, and derivation of postconditions for imperative programs.

Finally, we develop an infinitary calculus for DFOL plus iteration, with a completeness proof. Details of the relationships with existing calculi are given below. The two calculi that are the subject of this paper form the computation and inference engine of a toy programming language for theorem proving and computing with DFOL, *Dynamo*.

2 Dynamic First Order Logic

Dynamic First Order Logic results from interpreting quantification over v as change of valuation over the v position, conjunction as sequential composition, disjunction as nondeterministic choice, and negation as (negated) test for continuation. See Groenendijk and Stokhof [18] for a presentation and Visser [33] for an in-depth analysis. A sound and complete sequent style calculus for DFOL (without choice) was presented in Van Eijck [14]. In this paper we present a calculus that also covers the choice operator, and that is much closer to standard analytic tableau style reasoning for FOL (see Smullyan [31] for a classical presentation, Fitting [15] for a textbook treatment and connections with automated theorem proving, [19] for an excellent overview, and [9] for an encyclopedic account).

For applications of DFOL to programming, the presence of the choice operation $\cup$ in the language is crucial: choice is the basis of 'if then else', and of all nondeterministic programming constructs for exploring various avenues towards a solution. It can (and has been) argued that the full expressive power of $\cup$ is not necessary for applications of DFOL to natural language semantics. In fact, the presentation of dynamic predicate logic (DPL) in [18] does not cover $\cup$: in DPL, choice is handled in terms of negation and conjunction, with the argument that natural language 'or' is externally static. This means that an 'or' construction behaves like a test. The present calculus deals with DFOL including choice.

A very convenient extension that we immediately add to DFOL is representation of simultaneous binding. It is well known that bindings or substitutions are definable in DFOL. Still we will consider them as operators in their own right, in the spirit of Venema [32], where substitutions are studied as modal operators. Simultaneous bindings can in general not be expressed in terms of single bindings without introducing auxiliary variables. E.g., the swap of variables x and y in the simultaneous binding $[y/x, x/y]$ can only be expressed as a sequence of single bindings at the expense of availing ourselves of an extra variable z, as $z := x; x := y; y := z$. The dynamic effect of this sequence of single bindings is not quite the same as that of $[y/x, x/y]$, for $z := x; x := y; y := z$ changes the value of z, while $[y/x, x/y]$ does not, and the semantics of DFOL is sensitive to such subtle differences.

A first order signature Σ is a pair $\langle P_\Sigma, F_\Sigma \rangle$, with P_Σ a set of predicate constants and F_Σ a set of function constants. Let V be an infinite set of variables, and let $a : (P_\Sigma \cup F_\Sigma) \to \mathbb{N}$ be a function that assigns to every predicate or function symbol its arity. The function symbols with arity 0 are the individual constants. The set T_Σ of terms over the signature is given in the familiar way, by $t ::= v \mid f t_1 \cdots t_n$, where v ranges over V and f over F_Σ, with $a(f) = n$. The sub-terms of a term are given as usual. We will write sequences of terms $t_1, \ldots, t_n$ as $\bar{t}$.

A binding θ is a function $V \to T_\Sigma$ that makes only a finite number of changes, i.e., θ has the property that $dom(\theta) = \{v \in V \mid \theta(v) \neq v\}$ is finite. See Apt [1] and

Doets [11] for lucid introductions to the subject of binding in the context of logic programming. We will use $rng(\theta)$ for $\{\theta(v) \in T_\Sigma \mid \theta(v) \neq v\}$, and $var(rng(\theta))$ for $\cup\{var(\theta(v)) \mid v \in dom(\theta)\}$, where $var(t)$ is the set of variables occurring as a subterm in t. An explicit form (or: a representation) for binding θ is a sequence

$$[\theta(v_1)/v_1, \ldots, \theta(v_n)/v_n],$$

where $\{v_1, \ldots, v_n\} = dom(\theta)$, (i.e., $\theta(v_i) \neq v_i$, for only the *changes* are listed), and $i \neq j$ implies $v_i \neq v_j$ (i.e., each variable in the domain is mentioned only once). We will use $[]$ for the binding that changes nothing, i.e, $[]$ is the only binding θ with $dom(\theta) = \emptyset$. We use θ, ρ, possibly with indices, as meta-variables ranging over bindings. Representations for bindings are given, as usual, by:

$$\theta \quad ::= \quad [] \mid [t_1/v_1, \ldots, t_n/v_n] \qquad \text{provided } t_i \neq v_i, \text{ and } v_i = v_j \text{ implies } i = j.$$

We let $\circ$ denote the syntactic operation of composition of binding representations:

> Let $\theta = [t_1/v_1, \ldots, t_n/v_n]$ and $\rho = [r_1/w_1, \ldots, r_m/w_m]$ be binding representations. Then $\theta \circ \rho$ is the result of removing from the sequence
>
> $$[\theta(r_1)/w_1, \ldots, \theta(r_m)/w_m, t_1/v_1, \ldots, t_n/v_n]$$
>
> the binding pairs $\theta(r_i)/w_i$ for which $\theta(r_i) = w_i$, and the binding pairs t_j/v_j for which $v_j \in \{w_1, \ldots, w_m\}$.

For example, $[x/y] \circ [y/z] = [x/z, x/y]$, $[x/z, y/x] \circ [z/x] = [x/z]$.

We are now in a position to define the DFOL language $\mathcal{L}_\Sigma$ over signature Σ. We distinguish between DFOL units and DFOL formulas (or sequences).

DEFINITION 2.1 (The DFOL language $\mathcal{L}_\Sigma$ over signature Σ)

$$\begin{array}{rcl} t & ::= & v \mid f\bar{t} \\ U & ::= & \theta \mid \exists v \mid P\bar{t} \mid t_1 \doteq t_2 \mid \neg(\phi) \mid (\phi_1 \cup \phi_2) \end{array}$$

We will omit parentheses where it doesn't create syntactic ambiguity, and allow the usual abbreviations: we write $\perp$ for $\neg([])$, $\neg P\bar{t}$ for $\neg(P\bar{t})$, $t_1 \neq t_2$ for $\neg(t_1 \doteq t_2)$, $\phi_1 \cup \phi_2$ for $(\phi_1 \cup \phi_2)$. Similarly, $(\phi \rightarrow \psi)$ abbreviates $\neg(\phi; \neg(\psi))$, $\forall v(\phi)$ abbreviates $\neg(\exists v; \neg(\phi))$. A formula ϕ is a literal if ϕ is of the form $P\bar{t}$ or $\neg P\bar{t}$, or of the form $t_1 \doteq t_2$ or $t_1 \neq t_2$. The complement $\overline{\phi}$ of a formula ϕ is given by: $\overline{\phi} := \psi$ if ϕ has the form $\neg(\psi)$ and $\overline{\phi} := \neg(\phi)$ otherwise. We abbreviate $\neg\neg(\phi)$ as $((\phi))$, and we will call formulas of the form $((\phi))$ *block* formulas.

We can think of formula ϕ as built up from units U by concatenation. For formula induction arguments, it is sometimes convenient to read a unit U as the formula $U; []$ (recall that $[]$ is the empty binding), thus using $[]$ for the empty list formula. In other words, we will silently add the $[]$ at the end of a formula list when we need its presence in recursive definitions or induction arguments on formula structure.

Given a first order model $\mathcal{M} = (D, I)$ for signature Σ, the semantics of DFOL language $\mathcal{L}_\Sigma$ is given as a binary relation on the set ${}^V D$, the set of all variable maps (variable states, valuations) into the domain of the model. We impose the usual non-empty domain constraint of FOL: any Σ model $\mathcal{M} = (D, I)$ has $D \neq \emptyset$. If $s, u \in {}^V D$, we use $s \sim_v u$ to indicate that s, u differ at most in their value for v, and $s \sim_X u$ to indicate that s, u differ at most in their values for the members of X. If $s \in {}^V D$ and $v, v' \in V$, we use $s[v'/v]$ for the valuation u given by $u(v) = s(v')$, and $u(w) = s(w)$ for all $w \in V$ with $w \neq v$. Also, if s and v are as before and $d \in D$ we use $s[d/v]$ for the valuation u given by $u(v) = d$, and $u(w) = s(w)$ for all $w \in V$ with $w \neq v$.

$\mathcal{M} \models_s P\bar{t}$ indicates that s satisfies the predicate $P\bar{t}$ in $\mathcal{M}$ according to the standard truth definition for classical first order logic. $[\![t]\!]_s^{\mathcal{M}}$ gives the denotation of t in $\mathcal{M}$ under s. If θ is a binding and s a valuation (a member of ${}^V D$), we will use s_θ for the valuation u given by $u(v) = [\![\theta(v)]\!]_s^{M}$.

Definition 2.2 (Semantics of DFOL)

$$
\begin{array}{rcl}
{}_s[\![\theta]\!]_u^{\mathcal{M}} & \text{iff} & u = s_\theta \\
{}_s[\![\exists v]\!]_u^{\mathcal{M}} & \text{iff} & s \sim_v u \\
{}_s[\![P\bar{t}]\!]_u^{\mathcal{M}} & \text{iff} & s = u \text{ and } \mathcal{M} \models_s P\bar{t} \\
{}_s[\![t_1 \doteq t_2]\!]_u^{\mathcal{M}} & \text{iff} & s = u \text{ and } [\![t_1]\!]_s^{\mathcal{M}} = [\![t_2]\!]_s^{\mathcal{M}} \\
{}_s[\![\neg(\phi)]\!]_u^{\mathcal{M}} & \text{iff} & s = u \text{ and there is no } u' \text{ with } {}_s[\![\phi]\!]_{u'}^{\mathcal{M}} \\
{}_s[\![\phi_1 \cup \phi_2]\!]_u^{\mathcal{M}} & \text{iff} & {}_s[\![\phi_1]\!]_u^{\mathcal{M}} \text{ or } {}_s[\![\phi_2]\!]_u^{\mathcal{M}} \\
{}_s[\![U;\phi]\!]_u^{\mathcal{M}} & \text{iff} & \text{there is a } u' \text{ with } {}_s[\![U]\!]_{u'}^{\mathcal{M}} \text{ and } {}_{u'}[\![\phi]\!]_u^{\mathcal{M}}
\end{array}
$$

Note that it follows from this definition that

$$
{}_s[\![(\!(\phi)\!)]\!]_u^{\mathcal{M}} \text{ iff } s = u \text{ and there is a } u' \text{ with } {}_s[\![\phi]\!]_{u'}^{\mathcal{M}}.
$$

Thus, block formulas have their dynamic effects blocked off: double negation transforms the semantic transition relation into a test.

We introduce a syntactic blocking operation on formulas as follows (= is used for syntactic identity):

Definition 2.3 (Blocking Operation on Formulas)

$$
\begin{array}{rcl}
(\theta)^{\boxdot} & := & (\!(\theta)\!) \\
(\exists v)^{\boxdot} & := & (\!(\exists v)\!) \\
(P\bar{t})^{\boxdot} & := & P\bar{t} \\
(t_1 \doteq t_2)^{\boxdot} & := & t_1 \doteq t_2 \\
(\neg(\phi))^{\boxdot} & := & \neg(\phi) \\
(\phi_1 \cup \phi_2)^{\boxdot} & := & \begin{cases} (\phi_1 \cup \phi_2) & \text{if } \phi_1{}^{\boxdot} = \phi_1, \phi_2{}^{\boxdot} = \phi_2, \\ (\!(\phi_1 \cup \phi_2)\!) & \text{otherwise} \end{cases} \\
(U;\phi)^{\boxdot} & := & \begin{cases} U;\phi & \text{if } U^{\boxdot} = U, \phi^{\boxdot} = \phi, \\ (\!(U;\phi)\!) & \text{otherwise.} \end{cases}
\end{array}
$$

E.g., $(\exists x; Px)^{\boxdot} = (\!(\exists x; Px)\!)$, and $(\neg(\exists x; Px))^{\boxdot} = \neg(\exists x; Px)$. By induction on formula structure we get from Definitions 2.2 and 2.3 that the blocking operation makes a formula into a test, in the following sense:

Proposition 2.4
For all $\mathcal{M}$ and all valuations s, u for $\mathcal{M}$, all $\mathcal{L}_\Sigma$ formulas ϕ: ${}_s[\![\phi^\boxdot]\!]_u^\mathcal{M}$ iff $s = u$ and there is a u' with ${}_s[\![\phi]\!]_{u'}^\mathcal{M}$.

The key relation we want to get to grips with in this paper is the dynamic entailment relation that is due to [18]:

Definition 2.5 (Entailment in DFOL)
ϕ dynamically entails ψ, notation $\phi \models \psi$, $:\Leftrightarrow$ for all $\mathcal{L}_\Sigma$ models $\mathcal{M}$, all valuations s, u for $\mathcal{M}$, if ${}_s[\![\phi]\!]_u^\mathcal{M}$ then there is a variable state u' for which ${}_u[\![\psi]\!]_{u'}^\mathcal{M}$.

3 Binding in DFOL

Bindings θ are lifted to (sequences of) terms and (sets of) formulas in the familiar way:

Definition 3.1 (Binding in DFOL)

$$\begin{array}{rcl}
\theta(ft_1 \cdots t_n) & := & f\theta(t_1)\cdots\theta(t_n) \\
\theta(t_1, \ldots, t_n) & := & \theta(t_1), \ldots, \theta(t_n) \\
\theta(\rho) & := & \theta \circ \rho \\
\theta(\rho; \phi) & := & (\theta \circ \rho)\phi \\
\theta(\exists v; \phi) & := & \exists v; \theta'\phi \text{ where } \theta' = \theta \backslash \{t/v \mid t \in T\} \\
\theta(P\bar{t}; \phi) & := & P\theta\bar{t}; \theta\phi \\
\theta(t_1 \doteq t_2; \phi) & := & \theta t_1 \doteq \theta t_2; \theta\phi \\
\theta((\phi_1 \cup \phi_2); \phi_3) & := & \theta(\phi_1; \phi_3) \cup \theta(\phi_2; \phi_3) \\
\theta(\neg(\phi_1); \phi_2) & := & \neg(\theta\phi_1); \theta\phi_2 \\
\theta(\{\phi_1, \ldots, \phi_n\}) & := & \{\theta(\phi_1), \ldots, \theta(\phi_n)\}
\end{array}$$

Note that it follows from this definition that

$$\theta((\!(\phi_1)\!); \phi_2) = (\!(\theta\phi_1)\!); \theta\phi_2.$$

Thus, binding distributes over block: this accounts for how $(\!(\cdots)\!)$ insulates dynamic binding effects.[1]

The composition $\theta \cdot \rho$ of two bindings θ and ρ has its usual meaning of 'θ after ρ', which we get by means of $\theta \cdot \rho(v) := \theta(\rho(v))$. It can be proved in the usual way, by induction on term structure, that the definition has the desired effect, in the sense that for all $t \in T$, for all binding representations θ, ρ: $(\theta \circ \rho)(t) = \theta(\rho(t)) = (\theta \cdot \rho)(t)$.

Here is an example of how to apply a binding to a formula:

$$\begin{array}{rl}
& [a/x]Px; (Qx \cup \exists x; \neg Px); Sx \\
= & Pa; [a/x](Qx \cup \exists x; \neg Px); Sx \\
= & Pa; ([a/x]Qx; Sx \cup [a/x]\exists x; \neg Px; Sx) \\
= & Pa; (Qa; Sa; [a/x] \cup \exists x; \neg Px; Sx)
\end{array}$$

[1] Our reasons, by the way, for preferring prefix notation for application of bindings over the more usual postfix notation have to do with the fact that in the rules of our calculus bindings have an effect on formulas on their right.

The binding definition for DFOL fleshes out what has been called the 'folklore idea in dynamic logic' (Van Benthem [7]) that syntactic binding $[t/v]$ works semantically as the program instruction $v := t$ (Goldblatt [17]), with semantics given by ${}_s[\![v := t]\!]_u^{\mathcal{M}}$ iff $u = s[[\![t]\!]_s^M/v]$. To see the connection, note that $v := t$ can be viewed as DFOL shorthand for $\exists v; v = t$, on the assumption that $v \notin var(t)$. To generalize this to the case where $v \in var(t)$ and to simultaneous binding, auxiliary variables must be used. The fact that we have simultaneous binding represented in the language saves us some bother about these.

In standard first order logic, sometimes it is not safe to apply a binding to a formula, because it leads to accidental capture of free variables. The same applies here. Applying binding $[x/y]$ to $\exists x; Rxy$ is not safe, as it would lead to accidental capture of the free variable y. The following definition defines safety of binding.

Definition 3.2 (Binding θ is safe for ϕ)

$$\begin{array}{rcl}
\theta \text{ is safe for } \rho & & \text{always} \\
\theta \text{ is safe for } \rho; \phi & :\Longleftrightarrow & \theta \circ \rho \text{ is safe for } \phi \\
\theta \text{ is safe for } P\bar{t}; \phi & :\Longleftrightarrow & \theta \text{ is safe for } \phi \\
\theta \text{ is safe for } t_1 \doteq t_2; \phi & :\Longleftrightarrow & \theta \text{ is safe for } \phi \\
\theta \text{ is safe for } \exists v; \phi & :\Longleftrightarrow & v \notin var(rng\,\theta') \text{ and } \theta' \text{ is safe for } \phi \\
 & & \quad \text{where } \theta' = \theta \backslash \{(v,t) \mid t \in T\} \\
\theta \text{ is safe for } \neg(\phi_1); \phi_2 & :\Longleftrightarrow & \theta \text{ is safe for } \phi_1 \text{ and } \theta \text{ is safe for } \phi_2 \\
\theta \text{ is safe for } (\phi_1 \cup \phi_2); \phi_3 & :\Longleftrightarrow & \theta \text{ is safe for } \phi_1; \phi_3 \text{ and } \theta \text{ is safe for } \phi_2; \phi_3
\end{array}$$

Note that there are ϕ with $[]$ not safe for ϕ. E.g., $[]$ is not safe for $[y/x]\exists y; Rxy$, because $[y/x]$ is not safe for $\exists y; Rxy$. The connection between syntactic binding and semantic assignment is formally spelled out in the following:

Lemma 3.3 (Binding Lemma for DFOL)
For all Σ models $\mathcal{M}$, all $\mathcal{M}$-valuations s, u, all $\mathcal{L}_\Sigma$ formulas ϕ, all bindings θ that are safe for ϕ:

$${}_s[\![\theta\phi]\!]_u^{\mathcal{M}} \text{ iff } {}_s[\![\theta; \phi]\!]_u^{\mathcal{M}}.$$

Proof. Induction on the structure of ϕ. ∎

Immediately from this we get the following:

Proposition 3.4
DFOL has greater expressive power than DFOL with quantification replaced by definite assignment $v := d$.

Proof. If ϕ is an $\mathcal{L}_\Sigma$ formula without quantifiers, every binding θ is safe for ϕ. By the binding lemma for DFOL, ϕ is equivalent to an $\mathcal{L}_\Sigma$ formula without quantifiers but with trailing bindings. It is not difficult to see that both satisfiability and validity of quantifier free $\mathcal{L}_\Sigma$ formulas with binding trails is decidable. ∎

In fact, the tableau system below constitutes a decision algorithm for satisfiability or validity of quantifier free $\mathcal{L}_\Sigma$ formulas, while the trailing bindings summarize the finite changes made to input valuations.

A comparison of our definition of binding for DFOL with that of Visser [33] and [34] reveals that Visser's notion of binding follows a different intuition, namely that binding in the empty formula yields the empty formula. We think our notion is more truly dynamic, as is witnessed by the fact that it allows us to prove a binding lemma in the presence of $\cup$, which Visser's notion does not.

In the calculus we will need $\mathit{input}(\phi)$, the set of variables that have an input constraining occurrence in ϕ (with $\phi \in \mathcal{L}_\Sigma$), Let $\mathit{var}(\bar{t})$ be the variables occurring in $\bar{t}$.

DEFINITION 3.5 (Input constrained variables of $\mathcal{L}_\Sigma$ formulas)

$$\begin{array}{rcl}
\mathit{input}(\theta) & := & \mathit{var}(\mathit{rng}(\theta)) \\
\mathit{input}(\theta;\phi) & := & \mathit{var}(\mathit{rng}(\theta)) \cup (\mathit{input}(\phi)\backslash \mathit{dom}(\theta)) \\
\mathit{input}(\exists v;\phi) & := & \mathit{input}(\phi)\backslash\{v\} \\
\mathit{input}(P\bar{t};\phi) & := & \mathit{var}(\bar{t}) \cup \mathit{input}(\phi) \\
\mathit{input}(t_1 \doteq t_2;\phi) & := & \mathit{var}\{t_1,t_2\} \cup \mathit{input}(\phi) \\
\mathit{input}(\neg(\phi_1);\phi_2) & := & \mathit{input}(\phi_1) \cup \mathit{input}(\phi_2) \\
\mathit{input}((\phi_1 \cup \phi_2);\phi_3) & := & \mathit{input}(\phi_1;\phi_3) \cup \mathit{input}(\phi_2;\phi_3).
\end{array}$$

The following proposition (the DFOL counterpart to the finiteness lemma from classical FOL) can be proved by induction on formula structure:

PROPOSITION 3.6
For all $\mathcal{L}_\Sigma$ models $\mathcal{M}$, all valuations s, s', u, u' for $\mathcal{M}$, all $\mathcal{L}_\Sigma$ formulas ϕ:

$${}_s[\![\phi]\!]_u^{\mathcal{M}} \text{ and } s \sim_{V\backslash \mathit{input}(\phi)} s' \text{ imply } \exists u' \text{ with } {}_{s'}[\![\phi]\!]_{u'}^{\mathcal{M}}.$$

4 Adaptation of Tableau Reasoning to a Dynamic Setting

In classical tableau theorem proving, when investigating whether ϕ logically implies ψ, one systematically explores possibilities to make ϕ true and ψ false. If all such explorations fail, we conclude that ψ does indeed follow from ϕ, if at least one exploration succeeds we have the makings of a counterexample, which can be read off from an open branch of a tableau in several ways (e.g., by making every fact on the true side of the tableau branch true in the model, and all other facts false, or by making every fact on the false side of the tableau false in the model, and all other facts true; see [6]).

In the course of dealing with the original ϕ and ψ we decompose them into parts, so in general the data structure we deal with in classical tableau proving has the form $\Phi \bullet \Psi$, where Φ, Ψ are finite sets of formulas, with Φ the formulas we are committed to making true and Ψ the formulas we are committed to making false. Instead of distinguishing between sets of true formulas Φ and sets of false formulas Ψ, we will use one-sided tableaux, with the rule for every operator o matched by a $\neg o$ rule.

The tableau rule for disjunction in classical logic illustrates this. A tableau splitting rule like $\vee$ has the node with the disjunction $\phi \vee \psi$ above the two branches with the disjuncts ϕ and ψ. The rule $\vee$ serves as the 'left-hand side rule', and is matched by a rule $\neg\vee$ for dealing with the 'right-hand side'.

$$\begin{tikzcd} & \phi \vee \psi \arrow[dl, no head] \arrow[dr, no head] & \\ \phi & & \psi \end{tikzcd} \qquad \begin{tikzcd} \neg(\phi \vee \psi) \arrow[d, no head] \\ \neg(\phi) \\[-20pt] \neg(\psi) \end{tikzcd}$$

In the dynamic version of FOL, order matters: the sequencing operator ';' is not commutative in general. Suppose Φ were to consist of $\exists x; Px$ and $\neg Px$. Then if we read Φ as $\exists x; Px; \neg Px$, we should get a contradiction, but if we read Φ as $\neg Px; \exists x; Px$ then the formula has a model that contains both Ps and non-Ps.

Suppose Φ were to consist of just $\exists x; Px; \neg(Qx \cup Sx)$. Then we can apply the $\neg\cup$ analogue of $\neg\vee$ to Φ, but we should make sure that the results of this application, $\neg Qx$ and $\neg Sx$, remain in the scope of $\exists x; Px$. In other words, the result should be: $\exists x; Px; \neg Qx; \neg Sx$, with both $\neg Qx$ and $\neg Sx$ in the dynamic scope of the quantifier $\exists x$. In the tableau calculus to be presented, we will ensure that negation rules $\neg o$ take dynamic context into account, and that all formulas come with an appropriate binding context, to be supplied by explicit bindings.

Local Bindings Versus Global Substitutions

We will only perform a binding θ on ϕ when needed; rather than compute $\theta\phi$, the tableau rules will store $\theta; \phi$, and compute the binding in single steps as the need arises. Tableau theorem proving can be viewed as a process of gradually building a domain D and working out requirements to be imposed on that domain. The tableau procedure that investigates whether ϕ dynamically implies ψ will build a domain with positive and negative facts. For this we employ an infinite set $F_{\mathbf{sko}}$ of skolem functions, with $F_{\mathbf{sko}} \cap F_\Sigma = \emptyset$, plus a set of fresh variables $\boldsymbol{X}$, with $V \cap \boldsymbol{X} = \emptyset$. Call the extended signature Σ^*, and the extended language $\mathcal{L}_{\Sigma^*}$. Let T_{Σ^*} be the terms of the extended language, and $T^V_{\Sigma^*}$ the terms of the extended language without occurrences of members of $\boldsymbol{X}$. Call these the *frozen* terms of $\mathcal{L}_{\Sigma^*}$, and bear in mind that frozen terms, unlike ground terms, may contain occurrences of variables in V. Call an $\mathcal{L}_{\Sigma^*}$ literal *frozen* if it contains only frozen terms.

The variables in $\boldsymbol{X}$ will function as universal tableau variables [15]. Where the bindings of the variables from V are local to a tableau branch, the bindings of the variables from $\boldsymbol{X}$ are global to the whole tableau. Next to the (local) bindings for the variables V of $\mathcal{L}_\Sigma$, we introduce (global) substitutions $\boldsymbol{\sigma}$ for the fresh variables $\boldsymbol{X}$ in $\mathcal{L}_{\Sigma^*}$, and extend these to (sequences of) terms and (sets of) formulas in the manner of Definition 3.1. A substitution $\boldsymbol{\sigma}$ is a unifier of a set of (sequences of) terms T if $\boldsymbol{\sigma}T$ contains a single term (sequence of terms). It is a most general unifier (MGU) of T if $\boldsymbol{\sigma}$ is a unifier of T, and for all unifiers $\boldsymbol{\rho}$ of T there is a $\boldsymbol{\theta}$ with $\boldsymbol{\sigma} = \boldsymbol{\theta} \cdot \boldsymbol{\rho}$. Similarly for formulas. Note that only unifiers for global *substitutions* (the term maps for the global tableau variables from $\boldsymbol{X}$) will ever be computed.

The definitions and results on binding extend to bindings with values in T_{Σ^*}, and to substitutions (domain $\subset \boldsymbol{X}$, values in T_{Σ^*}). Still, the global substitutions play an altogether different rôle in the tableau construction process, so we use a different notation for them, and write (representations for) global substitutions as

$$\{\boldsymbol{x}_1 \mapsto t_1, \ldots, \boldsymbol{x}_n \mapsto t_n\}.$$

5 Tableaux for DFOL Formula Sets

If Σ is a first order signature, a DFOL tableau over Σ is a finitely branching tree with nodes consisting of (sets of) $\mathcal{L}_{\Sigma^*}$ formulas. A branch in a tableau $\boldsymbol{T}$ is a maximal path in $\boldsymbol{T}$. We will follow custom in occasionally identifying a branch B with the set of its formulas.

Let Φ be a set of $\mathcal{L}_\Sigma$ formulas. A DFOL tableau for Φ is constructed by a (possibly infinite) sequence of applications of the following rules:

Initialization The tree consisting of a single node [] is a tableau for Φ.

Binding Composition Suppose $\boldsymbol{T}$ is a tableau for Φ and B a branch in $\boldsymbol{T}$. Let $\phi \in B \cup \Phi$, let $\theta;\rho$ occur in ϕ, and let ϕ' be the result of replacing $\theta;\rho$ in ϕ by $\theta \circ \rho$. Then the tree $\boldsymbol{T}'$ constructed from $\boldsymbol{T}$ by extending B by ϕ' is a tableau for Φ.

Expansion Suppose $\boldsymbol{T}$ is a tableau for Φ and B a branch in $\boldsymbol{T}$. Let $\phi \in B \cup \Phi$. Then the tree $\boldsymbol{T}'$ constructed from $\boldsymbol{T}$ by extending B according to one of the tableau expansion rules, applied to ϕ, is a tableau for Φ.

Equality Replacement Suppose $\boldsymbol{T}$ is a tableau for Φ and B a branch in $\boldsymbol{T}$. Let $t_1 \doteq t_2 \in B \cup \Phi$ or $t_2 \doteq t_1 \in B \cup \Phi$, and $L(t_3) \in B \cup \Phi$, where L is a literal. Suppose t_1, t_3 are unifiable with MGU $\boldsymbol{\sigma}$. Then $\boldsymbol{T}'$ constructed from $\boldsymbol{T}$ by applying $\boldsymbol{\sigma}$ to all formulas in $\boldsymbol{T}$, and extending branch $\boldsymbol{\sigma}B$ with $L(\boldsymbol{\sigma}t_2)$ is a tableau for Φ.

Closure Suppose $\boldsymbol{T}$ is a tableau for Φ and B a branch in $\boldsymbol{T}$, and L, L' are literals in $B \cup \Phi$. If $L, \overline{L'}$ are unifiable with MGU $\boldsymbol{\sigma}$ then $\boldsymbol{T}'$ constructed from $\boldsymbol{T}$ by applying σ to all formulas in $\boldsymbol{T}$ is a tableau for Φ.

Any tableau branch can be thought of as a database Φ of formulas true on that branch. Because our databases may contain (negated) identities, we need some preliminaries in order to define closure of a tableau. When checking for closure, we may consider the parameters from V occurring in literals along a tableau branch as existentially quantified. Occurrence of Pv along branch B does *not* mean that everything has property P, but rather that the thing referred to as v has P. Thus, the V-variables occurring in literals can be taken as *names.* We can *freeze* the parameters from $\boldsymbol{X}$ by mapping them to fresh parameters from V. Applying a freezing substitution to a tableau replaces references to 'arbitrary objects' $\boldsymbol{x}$, $\boldsymbol{y}$, ... , by 'arbitrary names.' What this means is that we can determine closure of a branch B in terms of the *congruence closure* of the set of equalities occurring in a frozen image $\boldsymbol{\sigma}B$ of the branch. See [5], Chapter 4, for what follows about congruence closures.

If Φ is set of $\mathcal{L}_{\Sigma^*}$ formulas without parameters from $\boldsymbol{X}$, the congruence closure of Φ, notation $\approx_\Phi$, is the smallest congruence on T that contains all the equalities in Φ. In general, $\approx_\Phi$ will be infinite: if $a \doteq b$ is an equality in Φ, and f is a one-placed function symbol in the language, then $\approx_\Phi$ will contain $fa \doteq fb, ffa \doteq ffb, fffa \doteq fffb, \ldots$. Therefore, one uses congruence closure modulo some finite set instead.

Let S be the set of all sub-terms (not necessarily proper) of terms occurring in a literal in Φ. Then the congruence closure of Φ modulo S, notation $\mathrm{CC}_S(\Phi)$, is the finite set of equalities $\approx_\Phi \cap (S \times S)$. We can decide whether $t \doteq t'$ in $\mathrm{CC}_S(\Phi)$; [5] gives an algorithm for computing $\mathrm{CC}_S(G)$, for finite sets of equalities G and terms S, in polynomial time.

Definition 5.1
$t \approx t'$ is suspended in frozen $\mathcal{L}_{\Sigma^*}$ formula set Φ if $t \doteq t' \in \mathrm{CC}_S(\Phi)$, where S is the set of all sub-terms of terms occurring in literals in Φ. We extend this notation to sequences: $\bar{t} \approx \bar{t}'$ is suspended in Φ if $t_1 \approx t'_1, \ldots, t_n \approx t'_n$ are suspended in Φ.

A frozen $\mathcal{L}_{\Sigma^*}$ formula set Φ is closed if either $\neg(\theta) \in \Phi$ (recall that $\bot$ is an abbreviation for $\neg([])$), or for some $\bar{t} \approx \bar{t}'$ suspended in Φ we have $P\bar{t} \in \Phi$, $\neg P\bar{t}' \in \Phi$, or for a pair of terms t_1, t_2 with $t_1 \approx t_2$ suspended in Φ we have $t_1 \neq t_2 \in \Phi$.

A tableau $\boldsymbol{T}$ is *closed* if there is a freezing substitution $\boldsymbol{\sigma}$ of $\boldsymbol{T}$ such that each of its branches $\boldsymbol{\sigma}B$ is closed.

6 Tableau Expansion Rules

Note that we can take the form of any $\mathcal{L}_{\Sigma^*}$ formula to be $\theta; \phi$, by prefixing or suffixing $[]$ as the need arises. The tableau rules have the effect that bindings get pushed from left to right in the tableaux, and appear as computed results at the open end nodes.

Conjunctive Type Here are the rules for formulas of conjunctive type (type α in the Smullyan typology):

$$\begin{array}{c} \theta;\ P\bar{t};\ \phi \\ | \\ P\theta\bar{t} \\ \theta;\ \phi \end{array} \qquad \begin{array}{c} \theta;\ t_1 \doteq t_2;\ \phi \\ | \\ \theta t_1 \doteq \theta t_2 \\ \theta \circ [\theta t_i / v];\ \phi \\ \\ \text{where } \theta t_i = v \in V, i \in \{1,2\} \end{array} \qquad \begin{array}{c} \theta;\ t_1 \doteq t_2;\ \phi \\ | \\ \theta t_1 \doteq \theta t_2 \\ \theta;\ \phi \\ \\ \text{where } \theta t_i \notin V, i \in \{1,2\} \end{array}$$

$$\begin{array}{c} \neg(\theta;\ (\phi_1 \cup \phi_2);\ \phi_3) \\ | \\ \neg(\theta;\ \phi_1;\ \phi_3) \\ \neg(\theta;\ \phi_2;\ \phi_3) \end{array} \qquad \begin{array}{c} \theta;\ (\!(\phi_1)\!);\ \phi_2 \\ | \\ (\!(\theta;\ \phi_1)\!) \\ \theta;\ \phi_2 \end{array}$$

$$\begin{array}{c} \theta;\ \neg(\phi_1);\ \phi_2 \\ | \\ \neg(\theta;\ \phi_1) \\ \theta;\ \phi_2 \end{array}$$

Call the formula at the top node of a rule of this kind α and the formulas at the leaves α_1, α_2. To expand a tableau branch B by an α rule, extend B with both α_1 and α_2.

Disjunctive Type The rules for formulas of disjunctive type (Smullyan's type β):

$$\begin{tikzcd}[column sep=small] & \neg(\theta;\ P\bar{t};\ \phi) \arrow[dl, dash] \arrow[dr, dash] & \\ \neg P\theta\bar{t} & & \neg(\theta;\ \phi) \end{tikzcd} \qquad \begin{tikzcd}[column sep=small] & \neg(\theta;\ t_1 \doteq t_2;\ \phi) \arrow[dl, dash] \arrow[dr, dash] & \\ \theta t_1 \neq \theta t_2 & & \neg(\theta;\ \phi) \end{tikzcd} \qquad \begin{tikzcd}[column sep=small] & \theta;\ (\phi_1 \cup \phi_2);\ \phi_3 \arrow[dl, dash] \arrow[dr, dash] & \\ \theta;\phi_1;\phi_3 & & \theta;\phi_2;\phi_3 \end{tikzcd}$$

$$\begin{tikzcd}[column sep=small] & \neg(\theta;\ \neg(\phi_1);\ \phi_2) \arrow[dl, dash] \arrow[dr, dash] & \\ ((\theta;\ \phi_1)) & & ((\theta;\ \phi_2)) \end{tikzcd}$$

Call the formula at the top node of a rule of this kind β, the formula at the left leaf β_1 and the formula at the right leaf β_2. To expand a tableau branch B by an β rule, either extend B with β_1 or with β_2.

Universal Type Rule for universal formulas (Smullyan's type γ):

$$\begin{tikzcd} \neg(\theta;\ \exists v;\ \phi) \arrow[d, dash] \\ \neg(\theta \circ [\boldsymbol{x}/v];\ \phi) \end{tikzcd}$$

Here $\boldsymbol{x}$ is a universal variable taken from $\boldsymbol{X}$ that is new to the tableau. Call the formula at the top node of a rule of this kind $\gamma(v)$, and the formula at the leaf γ_1. To expand a tableau branch B by an γ rule, extend B with γ_1.

Existential Type Rule for existential formulas (Smullyan's type δ):

$$\begin{tikzcd} \theta;\ \exists v;\ \phi \arrow[d, dash] \\ \theta \circ [\mathrm{sk}_{\theta;\exists v;\phi}(\boldsymbol{x}_1, \dots, \boldsymbol{x}_n)/v];\ \phi \end{tikzcd}$$

Here $\boldsymbol{x}_1, \dots, \boldsymbol{x}_n$ are the universal parameters upon which interpretation of $\exists v; \phi$ depends, and $\mathrm{sk}_{\theta;\exists v;\phi}(\boldsymbol{x}_1, \dots, \boldsymbol{x}_n)$ is a skolem constant that is new to the tableau branch.[2]

By Proposition 3.6, $\{\boldsymbol{x}_1, \dots, \boldsymbol{x}_n\}$ is a subset of $\mathit{input}(\theta; \exists v; \phi)$, or, since no members of $\boldsymbol{X}$ occur in ϕ or in $\mathit{dom}(\theta)$, a subset of $\boldsymbol{X} \cap \mathit{input}(\theta) = \boldsymbol{X} \cap \mathit{var}(\mathit{rng}(\theta))$. From this set, we only need[3]

$$\{\boldsymbol{x}_1, \dots, \boldsymbol{x}_n\} := \boldsymbol{X} \cap \mathit{var}(\mathit{rng}(\theta \restriction (\mathit{input}(\phi) \backslash \{v\}))).$$

Call the formula at the top node of a rule of this kind $\delta(v)$, and the formula at the leaf δ_1. To expand a tableau branch B by an δ rule, extend B with δ_1.

[2] It is well-known that this can be optimized so that the choice of skolem constant only depends on $\theta;\ \exists v;\ \phi$.

[3] In an implementation, it may be more efficient to not bother about computing $\mathit{input}(\phi)$, and instead work with $\{\boldsymbol{x}_1, \dots, \boldsymbol{x}_n\} := \boldsymbol{X} \cap \mathit{var}(\mathit{rng}(\theta))$.

Protected Versions of the Rules All of the rules above have protected versions, i.e., versions with the formula ϕ to which the rule applies of the form $\psi^{\boxdot}$. The blocking operator is inherited by all the daughter formulas. As an example, here are the protected versions of one of the conjunctive and one of the disjunctive rules:

$$\begin{array}{c} (\theta; P\bar{t}; \phi)^{\boxdot} \\ | \\ (P\theta\bar{t})^{\boxdot} \\ (\theta; \phi)^{\boxdot} \end{array} \qquad \begin{array}{cc} \multicolumn{2}{c}{(\theta; (\phi_1 \cup \phi_2); \phi_3)^{\boxdot}} \\ / & \backslash \\ (\theta; \phi_1; \phi_3)^{\boxdot} & (\theta; \phi_2; \phi_3)^{\boxdot} \end{array}$$

Applying Definition 2.3, we see that this boils down to the following:

$$\begin{array}{c} (\!(\theta; P\bar{t}; \phi)\!) \\ | \\ P\theta\bar{t} \\ (\!(\theta; \phi)\!) \end{array} \qquad \begin{array}{cc} \multicolumn{2}{c}{(\!(\theta; (\phi_1 \cup \phi_2); \phi_3)\!)} \\ / & \backslash \\ (\!(\theta; \phi_1; \phi_3)\!) & (\!(\theta; \phi_2; \phi_3)\!) \end{array}$$

The tableau calculus specifies guidelines for extending a tableau tree with new leaf nodes. If one starts out from a single formula, at each stage only a finite number of rules can be applied. Breadth first search will get us all the possible tableau developments for a given initial formula, but this procedure is not an *algorithm*, for tableau proof construction: as in the tableau systems for classical FOL, there is no guarantee of termination.

7 Soundness of the Tableau Calculus

Valuations for Σ^* models $\mathcal{M} = (D, I)$ are functions in $V \cup \boldsymbol{X} \to D$. Any such function g can be viewed as a union $s \cup h$ of a function $s \in V \to D$ and a function $h \in \boldsymbol{X} \to D$ (take $s = g \upharpoonright V$ and $h = g \upharpoonright \boldsymbol{X}$). For satisfaction in Σ^* models we use the notation ${}_{s\cup h}[\![\phi]\!]^{M}_{u}$, to be understood in the obvious way. In terms of this we define the notion that we need to account for the universal nature of the $\boldsymbol{X}$ variables.

Definition 7.1

Let $\phi \in \mathcal{L}_{\Sigma^*}$, $\mathcal{M} = (D, I)$ a Σ^* model, $s, u \in V \to D$.

Then ${}^{\forall}_{s}[\![\phi]\!]^{\mathcal{M}}$ iff for every $h : \boldsymbol{X} \to D$ there is a $u : V \cup \boldsymbol{X} \to D$ with ${}_{s\cup h}[\![\phi]\!]^{\mathcal{M}}_{u}$. We say: s universally satisfies ϕ in $\mathcal{M}$.

For any tableau $\boldsymbol{T}$ we say that $\mathbf{C}(\boldsymbol{T})$ if there is an Σ^* model $\mathcal{M}$, a branch B of $\boldsymbol{T}$ and a V valuation s for $\mathcal{M}$ such that every formula ϕ of B is universally satisfied by s in $\mathcal{M}$.

Lemma 7.2

If s universally satisfies ϕ in $\mathcal{M}$, and $\boldsymbol{\sigma}$ is a substitution on $\boldsymbol{X}$ that is safe for ϕ, then s universally satisfies $\boldsymbol{\sigma}\phi$ in $\mathcal{M}$.

Proof. If ${}^{\forall}_{s}[\![\phi]\!]^{\mathcal{M}}$ then for every $\boldsymbol{X}$ valuation h in $\mathcal{M}$ there is a $V \cup \boldsymbol{X}$ valuation u in $\mathcal{M}$ with ${}_{s\cup h}[\![\phi]\!]^{\mathcal{M}}_{u}$. Thus for every h in $\mathcal{M}$ there is a $V \cup \boldsymbol{X}$ valuation u in $\mathcal{M}$ with

$${}_{s\cup h_{\boldsymbol{\sigma}}}[\![\phi]\!]^{\mathcal{M}}_{u},$$

and therefore for every h in $\mathcal{M}$ there is a $V \cup \boldsymbol{X}$ valuation u in $\mathcal{M}$ with

$${}_{s\cup h}[\![\boldsymbol{\sigma}; \phi]\!]^{\mathcal{M}}_{u}.$$

Since $\boldsymbol{\sigma}$ is safe for ϕ we have by the binding lemma that $[\![\boldsymbol{\sigma}\phi]\!]^{\mathcal{M}} = [\![\boldsymbol{\sigma};\phi]\!]^{\mathcal{M}}$, and it follows that s universally satisfies $\boldsymbol{\sigma}\phi$ in $\mathcal{M}$. ∎

With this, we can show that the tableau building rules preserve the $\mathbf{C}(\boldsymbol{T})$ relation.

Lemma 7.3 (Tableau Expansion Lemma)

1. If tableau $\boldsymbol{T}$ for Φ yields tableau $\boldsymbol{T}'$ by an application of binding composition, then $\mathbf{C}(\boldsymbol{T})$ implies $\mathbf{C}(\boldsymbol{T}')$.
2. If tableau $\boldsymbol{T}$ for Φ yields tableau $\boldsymbol{T}'$ by an application of a tableau expansion rule, then $\mathbf{C}(\boldsymbol{T})$ implies $\mathbf{C}(\boldsymbol{T}')$.
3. If tableau $\boldsymbol{T}$ for Φ yields tableau $\boldsymbol{T}'$ by an application of equality replacement, then $\mathbf{C}(\boldsymbol{T})$ implies $\mathbf{C}(\boldsymbol{T}')$.
4. If tableau $\boldsymbol{T}$ for Φ yields tableau $\boldsymbol{T}'$ by an application of closure, then $\mathbf{C}(\boldsymbol{T})$ implies $\mathbf{C}(\boldsymbol{T}')$.

Proof. 1. Immediate from the fact that $\theta;\rho$ and $\theta \circ \rho$ have the same interpretation.

2. All of the α and β rules are straightforward, except perhaps for the α equality rules. The change of θ to $\theta \circ [\theta t_i/v]$, where $\theta t_j = v$ $(i,j \in \{1,2\}, i \neq j,)$ reflects the fact that $\theta t_1 \doteq \theta t_2$ gives us the information to instantiate v.

The γ rule. Assume $\neg(\theta;\exists v;\phi)$ is universally satisfied by s in $\mathcal{M}$. We may assume that θ is safe for $\exists v;\phi$. If $\boldsymbol{x} \in \boldsymbol{X}$, $\boldsymbol{x}$ fresh to the tableau, then $\theta \circ [\boldsymbol{x}/v]$ will be safe for ϕ, and $\neg(\theta \circ [\boldsymbol{x}/v];\phi)$ will be universally satisfied by s in $\mathcal{M}$.

The δ rule. Assume s universally satisfies $\theta;\exists v;\phi$ in $\mathcal{M}$. By induction on tableau structure, $dom(\theta) \subset V$. Define a new model $\mathcal{M}'$ where $\mathrm{sk}_{\theta;\exists v;\phi}$ is interpreted as the function $f : D^n \to D$ given by $f(d_1, \dots, d_n) :=$ some d for which ϕ succeeds in $\mathcal{M}$ for input state $s_\theta[d_1/\boldsymbol{x}_1, \dots, d_n/\boldsymbol{x}_n, d/v]$. By the fact that s universally satisfies $\theta;\exists v;\phi$ in $\mathcal{M}$ and by the way we have picked $\boldsymbol{x}_1, \dots, \boldsymbol{x}_n$, such a d must exist. Then s will universally satisfy $\theta \circ [\mathrm{sk}_{\theta;\exists v;\phi}(\boldsymbol{x}_1, \dots, \boldsymbol{x}_n)/v];\phi$ in $\mathcal{M}'$, while universal satisfaction of other formulas on the branch is not affected by the switch from $\mathcal{M}$ to $\mathcal{M}'$.

3 and 4 follow immediately from Lemma 7.2. ∎

Theorem 7.4 (Soundness)
If $\phi, \psi \in \mathcal{L}_\Sigma$, and the tableau for $\phi;\neg(\psi)$ closes, then $\phi \models \psi$.

Proof. If the tableau for $\phi;\neg(\psi)$ closes, then by the Tableau Expansion Lemma, there are no $\mathcal{M}, s$ such that $\overset{\forall}{_s}[\![\phi;\neg(\psi)]\!]^{\mathcal{M}}$. Since $\phi, \psi \in \mathcal{L}_\Sigma$, there are no $\mathcal{M}, s, u$ with ${}_s[\![\phi;\neg(\psi)]\!]_u^{\mathcal{M}}$. In other words, for every Σ model $\mathcal{M}$ and every pair of variable states s, u for $\mathcal{M}$ with ${}_s[\![\phi]\!]_u^{\mathcal{M}}$ there has to be a variable state u' with ${}_u[\![\psi]\!]_{u'}^{\mathcal{M}}$. Thus, we have $\phi \models \psi$ in the sense of Definition 2.5. ∎

8 Derived Principles

Universal Quantification Immediately from the definition of $\forall v(\phi)$ we get:

$$\begin{array}{c} \theta;\forall v(\phi_1);\phi_2 \\ | \\ ((\theta \circ [\boldsymbol{x}/v];\phi_1)) \\ \theta;\phi_2 \end{array}$$

where $\boldsymbol{x} \in \boldsymbol{X}$ new to the tableau

Blocks Detachment A sequence of blocks $\pm(\phi_1);\dots;\pm(\phi_n)$, where $\pm(\phi_i)$ is either $(\!(\phi_i)\!)$ or $\neg(\phi_i)$, yields the set of its components, by a series of applications of distribution of the empty substitution over block or negation. This is useful, as the formulas $\pm(\phi_1),\dots,\pm(\phi_n)$ can be processed in any order. In a schema:

$$\begin{array}{c} \pm(\phi_1);\dots;\pm(\phi_n) \\ | \\ \pm(\phi_1) \\ \vdots \\ \pm(\phi_n) \end{array}$$

Negation Splitting The following rules are admissible in the calculus:

$$\begin{array}{c} \neg(\phi;\neg(\psi);\chi) \\ \diagup \qquad \diagdown \\ (\!(\phi;\psi)\!) \qquad \neg(\phi;\chi) \end{array} \qquad\qquad \begin{array}{c} \neg(\phi;(\!(\psi)\!);\chi) \\ \diagup \qquad \diagdown \\ (\!(\phi;\neg(\psi))\!) \qquad \neg(\phi;\chi) \end{array}$$

Negation splitting can be viewed as the DFOL guise of a well known principle from modal logic: $\Box(A \vee B) \rightarrow (\Diamond A \vee \Box B)$. To see the connection, note that $\neg(\phi;\neg(\psi);\chi)$ is semantically equivalent to $\neg(\phi;\neg(\psi \cup \neg(\chi)))$, where $\neg(\phi;\neg\cdots)$ behaves as a $\Box$ modality.

9 Examples

In the examples we will use $v_0, v_1, \dots$ as 0-ary skolem terms for v, etcetera.

Syllogistic Reasoning Consider the syllogism:

$$\forall x(Ax \rightarrow Bx), \forall x(Bx \rightarrow Cx) \models \forall x(Ax \rightarrow Cx).$$

This is an abbreviation of (9.1).

$$\neg(\exists x; Ax; \neg Bx), \neg(\exists x; Bx; \neg Cx) \;\models\; \neg(\exists x; Ax; \neg Cx) \tag{9.1}$$

The DFOL tableau for this example, a tableau refutation of

$$\neg(\exists x; Ax; \neg Bx); \neg(\exists x; Bx; \neg Cx); (\!(\exists x; Ax; \neg Cx)\!)$$

is in Figure 1.

Dynamic Donkey Reasoning The hackneyed example for dynamic binding in natural language, *If a farmer owns a donkey, he beats it*, has the following DFOL shape:

$$(\exists x; \exists y; Fx; Dy; Oxy \rightarrow Bxy),$$

which is shorthand for:

$$\neg(\exists x; \exists y; Fx; Dy; Oxy; \neg Bxy).$$

Consider the natural language text in (9.2).

If a farmer owns a donkey, he beats it. *A.* is a farmer and owns a donkey. (9.2)

FIG. 1. DFOL Tableau for Syllogistic Reasoning (9.1).

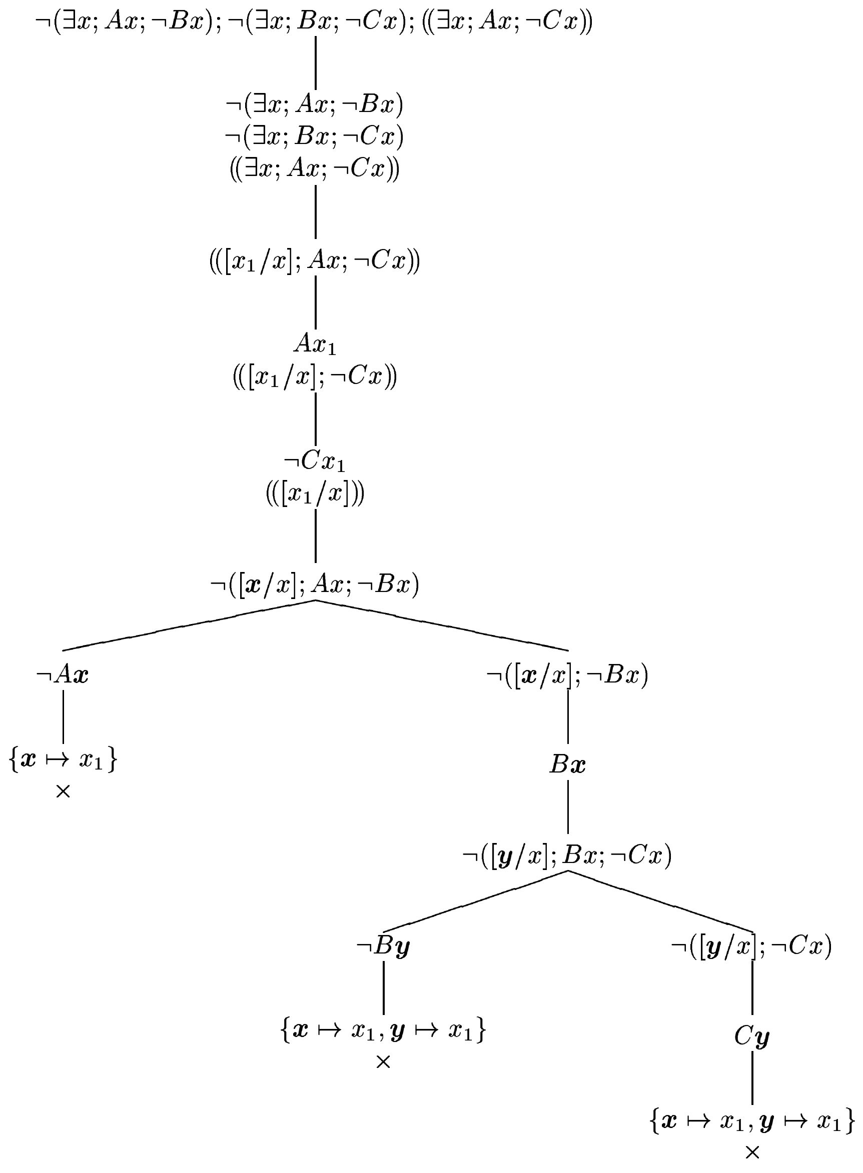

$\neg(\exists x; Ax; \neg Bx); \neg(\exists x; Bx; \neg Cx); ((\exists x; Ax; \neg Cx))$
$\neg(\exists x; Ax; \neg Bx)$
$\neg(\exists x; Bx; \neg Cx)$
$((\exists x; Ax; \neg Cx))$
$(([x_1/x]; Ax; \neg Cx))$
Ax_1
$(([x_1/x]; \neg Cx))$
$\neg Cx_1$
$(([x_1/x]))$
$\neg([\boldsymbol{x}/x]; Ax; \neg Bx)$
$\neg A\boldsymbol{x}$
$\{\boldsymbol{x} \mapsto x_1\}$
$\times$
$\neg([\boldsymbol{x}/x]; \neg Bx)$
$B\boldsymbol{x}$
$\neg([\boldsymbol{y}/x]; Bx; \neg Cx)$
$\neg B\boldsymbol{y}$
$\{\boldsymbol{x} \mapsto x_1, \boldsymbol{y} \mapsto x_1\}$
$\times$
$\neg([\boldsymbol{y}/x]; \neg Cx)$
$C\boldsymbol{y}$
$\{\boldsymbol{x} \mapsto x_1, \boldsymbol{y} \mapsto x_1\}$
$\times$

FIG. 2. Tableau for Dynamic Donkey Reasoning (9.2).

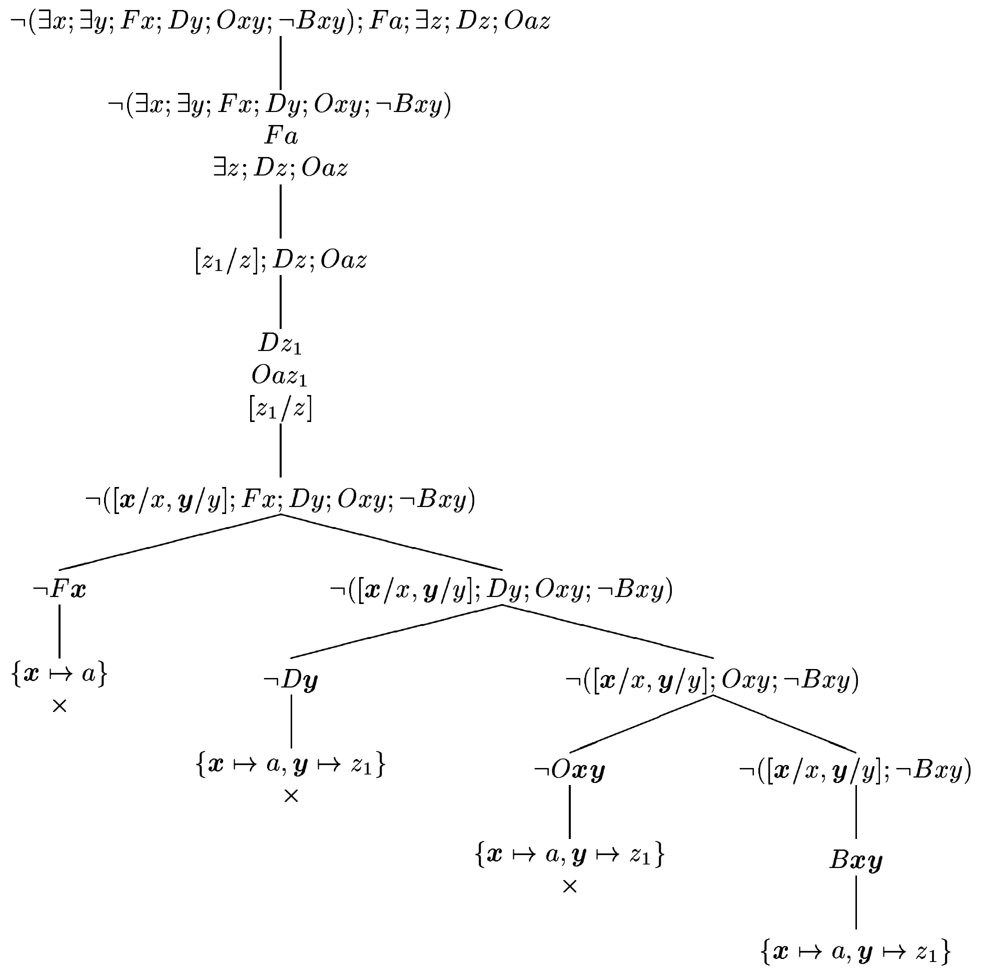

$\neg(\exists x; \exists y; Fx; Dy; Oxy; \neg Bxy); Fa; \exists z; Dz; Oaz$
$\neg(\exists x; \exists y; Fx; Dy; Oxy; \neg Bxy)$
Fa
$\exists z; Dz; Oaz$
$[z_1/z]; Dz; Oaz$
Dz_1
Oaz_1
$[z_1/z]$
$\neg([\boldsymbol{x}/x, \boldsymbol{y}/y]; Fx; Dy; Oxy; \neg Bxy)$
$\neg F\boldsymbol{x}$
$\neg([\boldsymbol{x}/x, \boldsymbol{y}/y]; Dy; Oxy; \neg Bxy)$
$\{\boldsymbol{x} \mapsto a\}$
$\times$
$\neg D\boldsymbol{y}$
$\neg([\boldsymbol{x}/x, \boldsymbol{y}/y]; Oxy; \neg Bxy)$
$\{\boldsymbol{x} \mapsto a, \boldsymbol{y} \mapsto z_1\}$
$\times$
$\neg O\boldsymbol{xy}$
$\neg([\boldsymbol{x}/x, \boldsymbol{y}/y]; \neg Bxy)$
$\{\boldsymbol{x} \mapsto a, \boldsymbol{y} \mapsto z_1\}$
$\times$
$B\boldsymbol{xy}$
$\{\boldsymbol{x} \mapsto a, \boldsymbol{y} \mapsto z_1\}$

Figure 2 shows how to draw conclusions from the DFOL version of this text in a DFOL tableau calculation.

The open tableau branch in Figure 2 yields the fact Baz_1, plus the following further information about z_1: Dz_1, Oaz_1. This further information is useful to identify z_1 as *the donkey that Alfonso owns* (or perhaps *a donkey that Alfonso owns*) that was introduced in the text.

Open Tableau Branches, Partial Models, Reference Resolution An open tableau branch for a DFOL formula ϕ may be viewed as a *partial model* for ϕ, with just enough information to verify the formula. For instance, the open branch in the previous example does not specify whether donkey z_1 also beats Alfonso or not: Bz_1a is neither among the facts (true atoms) nor among the negated facts (false atoms) of the branch.

In tableau branches involving equality there is also another kind of partiality involved: the terms are ***proto-objects*** rather than genuine objects, in sense that they have not yet 'made up their minds' about which individual they are: two terms t_1, t_2 on a tableau that does not contain $t_1 \neq t_2$ may be interpreted as a single individual. This is because the information about equality that the branch provides is also partial. Also, variables from $\boldsymbol{X}$ (free tableau variables) can be resolved to any object whatsoever.

The level of tableau style generation of partial models for discourse may be just the right level for pronoun reference resolution (cf. the suggestion in [8]). Since reference resolution is a processing step that links a pronoun to a suitable antecedent, what about equating the suitable antecedents with the available terms of the branches in a tableau? After all, reference resolution for pronouns is part of semantic processing, so it has a more natural habitat at the level of processing NL representations than at the level of mere representation of NL meaning.

Building on this idea, we (tentatively) introduce the following rule for pronoun resolution:

$$\begin{array}{c} P\text{pro} \\ | \\ Pt \\ \\ t \text{ occurs on the branch} \end{array} \qquad\qquad \begin{array}{c} \neg P\text{pro} \\ | \\ \neg Pt \\ \\ t \text{ occurs on the branch} \end{array}$$

Of course, for a full account one would need rules to determine the *salient* terms for pronoun resolution along a branch, but here we will just demonstrate the rule with a tableau for the following piece of discourse.

Every farmer owns a donkey. Some farmer beats it. (9.3)

See Figure 3. Intuitively, in this tableau, the following happens. First, a term z_1 introduced for *Some farmer*. This leads to an unresolved fact '$B(z_1, \text{it})$' in the database of the partial model under construction. Later, the pronoun *it* is resolved to 'the donkey that z_1 owns' generated from *every farmer owns a donkey*, and represented in the database of the partial model as $\text{sk}_1(z_1)$.

Here is another well-known example from the literature that is hard to crack in a purely representational setting (a piece of evidence against the claim, by the way, that 'or' in natural language is externally static):

John owns a motorbike or a car. It is in the garage. (9.4)

Again, in the tableau setting there is no problem: the tableau for (9.4) will have two branches, and both of the branches will contain a suitable antecedent for *it.*

Reasoning about '$<$' Consider example (9.5).

$$y < x; \neg(\exists x; \exists y; x < y). \tag{9.5}$$

This is contradictory, for first two objects of different size are introduced, and next we are told that all objects have the same size. The contradiction is derived as follows:

FIG. 3. Tableau for Donkey Reasoning with Pronoun Resolution (9.3).

$$\begin{tikzcd}
 & \forall x(Fx \rightarrow \exists y; Dy; Oxy); \exists z; Fz; B(z, \mathrm{it}) \arrow[d, dash] & \\
 & \begin{array}{c} \forall x(Fx \rightarrow \exists y; Dy; Oxy) \\ \exists z; Fz; B(z, \mathrm{it}) \end{array} \arrow[d, dash] & \\
 & {[z_1/z]; Fz; B(z, \mathrm{it})} \arrow[d, dash] & \\
 & \begin{array}{c} Fz_1 \\ {[z_1/z]; B(z, \mathrm{it})} \end{array} \arrow[d, dash] & \\
 & B(z, \mathrm{it}) \arrow[d, dash] & \\
 & \neg(\exists x; Fx; \neg(\exists y; Dy; Oxy)) \arrow[d, dash] & \\
 & \neg([\boldsymbol{x}/x]; Fx; \neg(\exists y; Dy; Oxy)) \arrow[dl, dash] \arrow[dr, dash] & \\
\neg F\boldsymbol{x} \arrow[d, dash] & & (([\boldsymbol{x}/x]; \exists y; Dy; Oxy)) \arrow[d, dash] \\
\begin{array}{c} \{\boldsymbol{x} \mapsto z_1\} \\ \times \end{array} & & (([\boldsymbol{x}/x, \mathrm{sk}_1(\boldsymbol{x})/y]; Dy; Oxy)) \arrow[d, dash] \\
 & & \begin{array}{c} D\mathrm{sk}_1(\boldsymbol{x}) \\ (([\boldsymbol{x}/x, \mathrm{sk}_1(\boldsymbol{x})/y]; Oxy)) \end{array} \arrow[d, dash] \\
 & & O(\boldsymbol{x}, \mathrm{sk}_1(\boldsymbol{x})) \arrow[d, dash] \\
 & & B(z_1, \mathrm{sk}_1(\boldsymbol{x})) \arrow[d, dash] \\
 & & \{\boldsymbol{x} \mapsto z_1\}
\end{tikzcd}$$

$$\begin{tikzcd}
y < x; \neg(\exists x; \exists y; x < y) \arrow[d, dash] \\
\begin{array}{c} y < x \\ \neg(\exists x; \exists y; x < y) \end{array} \arrow[d, dash] \\
\neg([\boldsymbol{x}_1/x, \boldsymbol{x}_2/y]; x < y) \arrow[d, dash] \\
\neg \boldsymbol{x}_1 < \boldsymbol{x}_2 \arrow[d, dash] \\
\begin{array}{c} \{\boldsymbol{x}_1 \mapsto y, \boldsymbol{x}_2 \mapsto x\} \\ \times \end{array}
\end{tikzcd}$$

Computation of Answer Substitutions The following example illustrates how the tableau calculus can be used to compute answer substitutions for a query.

$$\begin{tikzcd}
 & x < 3; x \doteq 5 \cup x \doteq 2 \arrow[d, dash] & \\
 & \begin{array}{c} x < 3 \\ x \doteq 5 \cup x \doteq 2 \end{array} \arrow[dl, dash] \arrow[dr, dash] & \\
x \doteq 5 \arrow[d, dash] & & x \doteq 2 \arrow[d, dash] \\
{[5/x]} & & {[2/x]}
\end{tikzcd}$$

A combination with model checking or term rewriting (see [12]) can be used to get rid of the left branch. Adding the relevant axioms for $<$ would achieve the same. See the next example.

More Reasoning about $<$ Assume that $1, 2, 3, \ldots$ are shorthand for $s0, ss0, sss0, \ldots$. We derive a contradiction from the assumption that $4 < 2$ together with two axioms for $<$. See Figure 4, with arrows connecting the literals that effect closure.

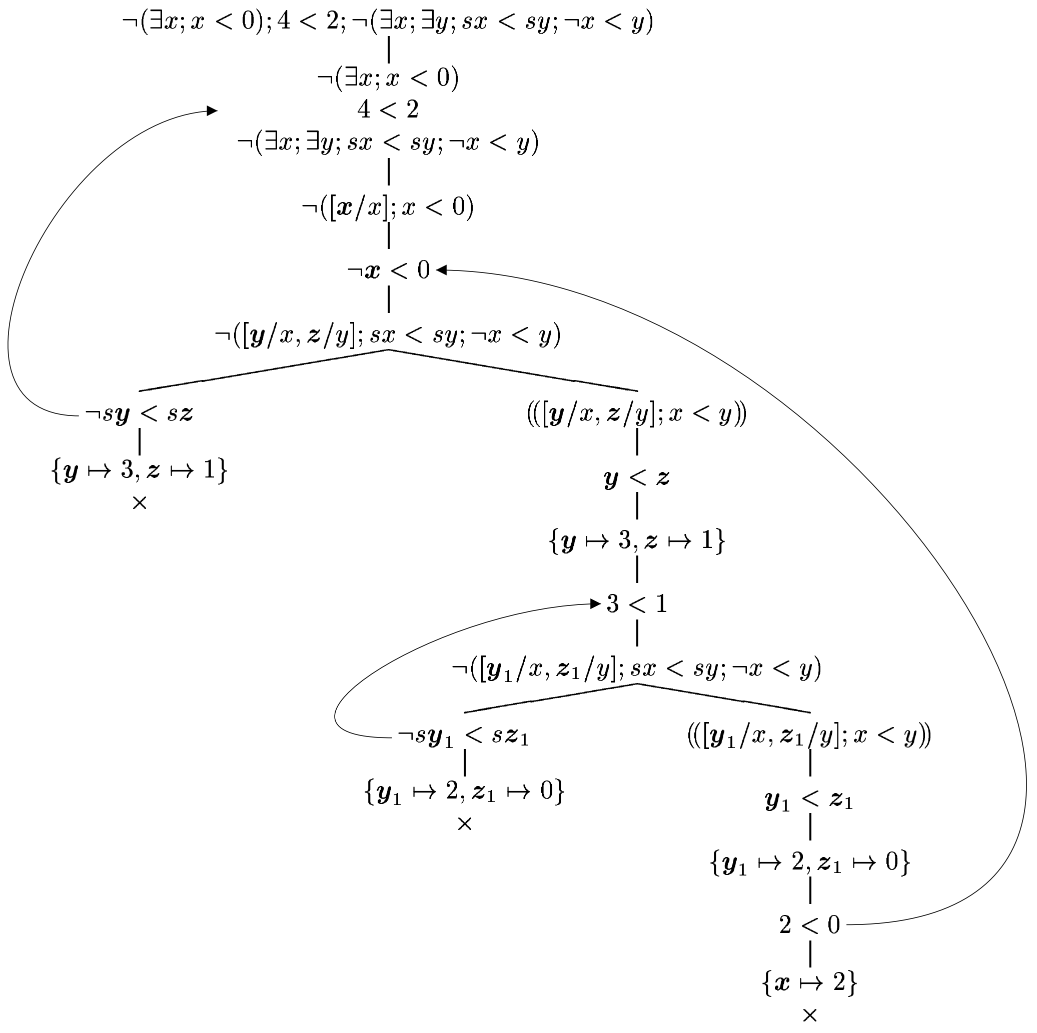
Computation of Answer Substitutions, with Variable Reuse Figure 5 demonstrates how the computed answer substitution stores the final value for x, under the renaming x_1. Because of the renaming, the database information for x_1 does not conflict with that for x.

Closure by Equality Replacement This example illustrates closure by means of e-quality replacement, in reasoning about $\exists x; \exists y; x \neq y; \exists x; \neg(\exists y; x \neq y)$. Note that x_1, y_1, x_2 serve as names for objects in the domain under construction. What the argument boils down to is: if the name x_2 applies to everything, then it cannot be the case that there are two different objects x_1, y_1. See Figure 6.

The first application of equality replacement in Figure 6 unifies $\boldsymbol{x}$ with x_1 and concludes from $x_2 \doteq \boldsymbol{x}, x_1 \neq y_1$ that $x_2 \neq y_1$. The second application of equality replacement unifies $\boldsymbol{y}$ with y_1 and concludes from $x_2 \doteq \boldsymbol{y}, x_2 \neq y_1$ that $x_2 \neq x_2$.

Loop Invariant Checking To check that $x = y!$ is a loop invariant for $y := y + 1; x := x * y$, assume it is not, and use the calculus to derive a contradiction with the definition of !. Note that $y := y + 1; x := x * y$ appears in our notation as $[y + 1/y]; [x * y/x]$. See Figure 7. A more detailed account would of course have to use the DFOL definitions

FIG. 4. More Reasoning about $<$.



$\neg(\exists x; x < 0); 4 < 2; \neg(\exists x; \exists y; sx < sy; \neg x < y)$
$\neg(\exists x; x < 0)$
$4 < 2$
$\neg(\exists x; \exists y; sx < sy; \neg x < y)$
$\neg([\boldsymbol{x}/x]; x < 0)$
$\neg \boldsymbol{x} < 0$
$\neg([\boldsymbol{y}/x, \boldsymbol{z}/y]; sx < sy; \neg x < y)$
$\neg s\boldsymbol{y} < s\boldsymbol{z}$
$\{\boldsymbol{y} \mapsto 3, \boldsymbol{z} \mapsto 1\}$
$\times$
$(([\boldsymbol{y}/x, \boldsymbol{z}/y]; x < y))$
$\boldsymbol{y} < \boldsymbol{z}$
$\{\boldsymbol{y} \mapsto 3, \boldsymbol{z} \mapsto 1\}$
$3 < 1$
$\neg([\boldsymbol{y}_1/x, \boldsymbol{z}_1/y]; sx < sy; \neg x < y)$
$\neg s\boldsymbol{y}_1 < s\boldsymbol{z}_1$
$\{\boldsymbol{y}_1 \mapsto 2, \boldsymbol{z}_1 \mapsto 0\}$
$\times$
$(([\boldsymbol{y}_1/x, \boldsymbol{z}_1/y]; x < y))$
$\boldsymbol{y}_1 < \boldsymbol{z}_1$
$\{\boldsymbol{y}_1 \mapsto 2, \boldsymbol{z}_1 \mapsto 0\}$
$2 < 0$
$\{\boldsymbol{x} \mapsto 2\}$
$\times$

of $+$, $*$ and !.

Loop Invariant Detection This time, we inspect the code $[x * (y + 1)/x]; [y + 1/y]$ starting from scratch. Since y is the variable that gets incremented, we may assume that x depends on y via an unknown function f. Thus, we start in a situation where $fy = x$. We check what has happened to this dependency after execution of the code $[x * (y + 1)/x]; [y + 1/y]$, by means of a tableau calculation for $fy \doteq x; [x * (y + 1)/x]; [y + 1/y]; fy \doteq x$. See Figure 8. The tableau shows that $[x * (y + 1)/x]; [y + 1/y]$ is a loop for the factorial function.

Postcondition Reasoning for 'If Then Else' For another example of this, consider a

FIG. 5. Computation of Answer Substitutions, with Variable Reuse

$$\begin{tikzcd}
& x \doteq 0; x \doteq y \cup y \doteq 2; \exists x; x \doteq 2 \arrow[d, dash] & \\
& \begin{array}{c} x \doteq 0 \\ {[0/x]}; x \doteq y \cup y \doteq 2; \exists x; x \doteq 2 \end{array} \arrow[dl, dash] \arrow[dr, dash] & \\
{[0/x]; x \doteq y; \exists x; x \doteq 2} \arrow[d, dash] & & {[0/x]; y \doteq 2; \exists x; x \doteq 2} \arrow[d, dash] \\
\begin{array}{c} 0 \doteq y \\ {[0/x, 0/y]}; \exists x; x \doteq 2 \end{array} \arrow[d, dash] & & \begin{array}{c} 2 \doteq y \\ {[0/x, 2/y]}; \exists x; x \doteq 2 \end{array} \arrow[d, dash] \\
{[x_1/x, 0/y]; x \doteq 2} \arrow[d, dash] & & {[x_1/x, 2/y]; x \doteq 2} \arrow[d, dash] \\
\begin{array}{c} x_1 \doteq 2 \\ {[x_1/x, 0/y, 2/x_1]} \end{array} & & \begin{array}{c} x_1 \doteq 2 \\ {[x_1/x, 2/y, 2/x_1]} \end{array}
\end{tikzcd}$$

loop through the following programming code:

$$i := i + 1; \text{if } x < a[i] \text{ then } x := a[i] \text{ else skip.} \tag{9.6}$$

Assume we know that before the loop x is the maximum of array elements $a[0]$ through $a[i]$. Then our calculus allows us to derive a characterization of the value of x at the end of the loop. Note that the loop code appears in DFOL under the following guise:

$$[i + 1/i]; (x < a[i]; [a[i]/x] \cup \neg x < a[i]).$$

The situation of x at the start of the loop can be given by an identity $x = m_i^0$, where m is a two-placed function. To get a characterization of x at the end, we just put $X = x$ (X a constant) at the end, and see what we get (Figure 9). What the leaf nodes tell us is that in any case, X is the maximum of $a[0], .., a[i + 1]$, and this maximum gets computed in x.

10 Completeness

Completeness for this calculus can be proved by a variation on completeness proofs for tableau calculi in classical FOL. First we define *trace sets* for DFOL as an analogue to Hintikka sets for FOL. A trace set is a set of DFOL formulas satisfying the closure conditions that can be read off from the tableau rules. Trace sets can be viewed as blow-by-blow accounts of particular consistent DFOL computation paths (i.e., paths that do not close).

DEFINITION 10.1
A set Ψ of $\mathcal{L}_{\Sigma^*}$ formulas is a **trace set** if the following hold:

FIG. 6. Reasoning With Equality

$$\begin{tikzcd}
\exists x; \exists y; x \neq y; \exists x; \neg\exists y; x \neq y \arrow[d, dash] \\
{[x_1/x, y_1/y]; x \neq y; \exists x; \neg\exists y; x \neq y} \arrow[d, dash] \\
\begin{array}{c} x_1 \neq y_1 \\ {[x_2/x, y_1/y]; \neg\exists y; x \neq y} \end{array} \arrow[d, dash] \\
\neg[x_2/x, \boldsymbol{x}/y]; x \neq y \arrow[d, dash] \\
x_2 \doteq \boldsymbol{x} \arrow[d, dash] \\
\begin{array}{c} \{\boldsymbol{x} \mapsto x_1\} \\ x_2 \neq y_1 \end{array} \arrow[d, dash] \arrow[uuu, bend left=60] \\
\neg[x_2/x, \boldsymbol{y}/y]; x \neq y \arrow[d, dash] \\
x_2 \doteq \boldsymbol{y} \arrow[d, dash] \\
\begin{array}{c} \{\boldsymbol{y} \mapsto y_1\} \\ x_2 \neq x_2 \\ \times \end{array} \arrow[uuu, bend right=60]
\end{tikzcd}$$

FIG. 7. Loop Invariant Checking.

$$\begin{tikzcd}
x = y!; [y+1/y]; [x * y/x]; x \neq y! \arrow[d, dash] \\
{[y!/x]; [y+1/y]; [x * y/x]; x \neq y!} \arrow[d, dash] \\
{[y!/x, y+1/y]; [x * y/x]; x \neq y!} \arrow[d, dash] \\
{[y+1/y, y! * (y+1)/x]; x \neq y!} \arrow[d, dash] \\
y! * (y+1) \neq (y+1)!
\end{tikzcd}$$

FIG. 8. Loop Invariant Detection.

$$\begin{tikzcd}
fy \doteq x; [x*(y+1)/x]; [y+1/y]; fy \doteq x \arrow[d, dash] \\
\begin{array}{c} fy \doteq x \\ {[fy/x]; [x*(y+1)/x]; [y+1/y]; fy \doteq x} \end{array} \arrow[d, dash] \\
{[fy*(y+1)/x]; [y+1/y]; fy \doteq x} \arrow[d, dash] \\
{[fy*(y+1)/x, y+1/y]; fy \doteq x} \arrow[d, dash] \\
\begin{array}{c} f(y+1) \doteq fy*(y+1) \\ {[fy*(y+1)/x, y+1/y]} \end{array}
\end{tikzcd}$$

FIG. 9. Postcondition Reasoning For (9.6).

$$\begin{tikzcd}
 & x = m_i^0; [i+1/i]; x < a[i]; [a[i]/x] \cup \neg x < a[i]; X = x \arrow[d, dash] & \\
 & {[m_i^0/x]; [i+1/i]; x < a[i]; [a[i]/x] \cup \neg x < a[i]; X = x} \arrow[d, dash] & \\
 & {[m_i^0/x, i+1/i]; x < a[i]; [a[i]/x] \cup \neg x < a[i]; X = x} \arrow[d, dash] & \\
 & {[m_i^0/x, i+1/i]; x < a[i]; [a[i]/x]; X = x \cup [m_i^0/x, i+1/i]; \neg x < a[i]; X = x} \arrow[dl, dash] \arrow[dr, dash] & \\
{[m_i^0/x, i+1/i]; x < a[i]; [a[i]/x]; X = x} \arrow[d, dash] & & {[m_i^0/x, i+1/i]; \neg x < a[i]; X = x} \arrow[d, dash] \\
\begin{array}{c} m_i^0 < a[i+1] \\ {[m_i^0/x, i+1/i]; [a[i]/x]; X = x} \end{array} \arrow[d, dash] & & \neg m_i^0 < a[i+1], [m_i^0/x, i+1/i]; X = x \arrow[d, dash] \\
{[i+1/i, a[i+1]/x]; X = x} \arrow[d, dash] & & \begin{array}{c} \neg m_i^0 < a[i+1] \\ X = m_i^0 \\ {[m_i^0/x, i+1/i]} \end{array} \\
\begin{array}{c} X = a[i+1] \\ {[i+1/i, a[i+1]/x]} \end{array} & &
\end{tikzcd}$$

1. $\neg(\theta) \notin \Psi$.
2. If $\phi \in \Psi$, then $\overline{\phi} \notin \Psi$.
3. If $\theta; \phi \in \Psi$, then $\theta\phi \in \Psi$.
4. If $\alpha \in \Psi$ then all $\alpha_i \in \Psi$.
5. If $\beta \in \Psi$ then at least one $\beta_i \in \Psi$.
6. If $\gamma(v) \in \Psi$, then $\gamma_1(t) \in \Psi$ for all $t \in T^V_{\Sigma^*}$ (all terms that do not contain variables from $\boldsymbol{X}$).
7. If $\delta(v) \in \Psi$, then $\delta_1(t) \in \Psi$ for some $t \in T^V_{\Sigma^*}$ (some term t that does not contain variables from $\boldsymbol{X}$).

This definition is motivated by the Trace Lemma:

LEMMA 10.2 (Trace Lemma)
The elements of every trace set Ψ are simultaneously satisfiable.

PROOF. Define a canonical model $\mathcal{M}_0$ in the standard fashion, using congruence closure on the trace set Ψ over the set of terms occurring in Φ, to get a suitable congruence $\equiv$ on terms. Next, define a canonical valuation s_0 by means of $s_0(v) := [v]_{\equiv}$ for members of V and $s_0(\mathrm{sk}^0_i) = [\mathrm{sk}^0_i]_{\equiv}$ for 0-ary skolem terms. Verify that s_o satisfies every member of Φ in $\mathcal{M}_0$. ∎

To employ the lemma, we need the standard notion of a fair computation rule. A computation rule is a function F that for any set of formulas Φ and any tableau $\boldsymbol{T}$, computes the next rule to be applied on $\boldsymbol{T}$. This defines a partial order on the set of tableaux for Φ, with the successor of $\boldsymbol{T}$ given by F. Then there is a (possibly infinite) sequence of tableaux for Φ starting from the initial tableau, and with supremum $\boldsymbol{T}_\infty$. A computation rule F is fair if the following holds for all branches B in $\boldsymbol{T}_\infty$:

1. All formulas of type α, β, δ occurring on B or in Φ were used to expand B,
2. All formulas of type γ occurring on B or in Φ were used infinitely often to expand B.

THEOREM 10.3 (Completeness)
For all $\phi, \psi \in \mathcal{L}_\Sigma$: if $\phi \models \psi$ then there is a tableau refutation of $\phi; \neg(\psi)$.

PROOF. Let $\boldsymbol{T}_0, \ldots$ be a sequence of tableaux for $\phi; \neg(\psi)$ constructed with a fair computation rule, without closure rule applications, and with supremum $\boldsymbol{T}_\infty$. Define a freezing map $\boldsymbol{\sigma}_\infty$ on $\boldsymbol{T}_\infty$ in the standard fashion (see, e.g., [19]). In particular, let $(B_k)_{k\geq 0}$ be an enumeration of the branches of $\boldsymbol{T}_\infty$, let $(\phi_i)_{i\geq 0}$ be an enumeration of the type γ formulas of $\boldsymbol{T}_\infty$, and let $\boldsymbol{x}_{ijk}$ be the variable introduced for the j-th application of γ formula ϕ_i along branch B_k. If $(t_j)_{j\geq 0}$ is an enumeration of all the frozen terms of $\boldsymbol{T}_\infty$, we can set $\boldsymbol{\sigma}_\infty(\boldsymbol{x}_{ijk}) := t_j$ for all $i, j, k \geq 0$. Note that $\boldsymbol{\sigma}_\infty$ is not, strictly speaking, a substitution since $dom(\boldsymbol{\sigma}_\infty)$ is not finite.

Suppose $\boldsymbol{\sigma}_\infty\boldsymbol{T}_\infty$ contains an open branch. Then from this branch we would get a trace set, which in turn would give a canonical model and a canonical valuation for $\phi; \neg(\psi)$, and contradiction with the assumption that $\phi \models \psi$. Therefore, $\boldsymbol{\sigma}_\infty\boldsymbol{T}_\infty$ must be closed.

Since the tree $\boldsymbol{T}_\infty$ is finitely branching and all formulas having an effect on closure are at finite distance from the root, there is a finite $\boldsymbol{T}_n$ with $\boldsymbol{\sigma}_\infty\boldsymbol{T}_n$ closed. Finally, construct an MGU $\boldsymbol{\sigma}$ for $\boldsymbol{T}_n$ on the basis of the part of $\boldsymbol{\sigma}_\infty$ that is actually used in the closure of $\boldsymbol{T}_n$, and we are done. ∎

THEOREM 10.4 (Computation Theorem)
If ϕ is satisfiable, then all bindings θ produced by open tableau branches B satisfy ${}_s[\![\phi]\!]^{\mathcal{M}}_{s_\theta}$, where $\mathcal{M}$ is the canonical model constructed from B, and s the canonical valuation.

PROOF. Let $\boldsymbol{T}_0, \ldots$ be a sequence of tableaux for ϕ constructed with a fair computation rule, without closure rule applications, and with supremum $\boldsymbol{T}_\infty$. Consider $\boldsymbol{\sigma}_\infty \boldsymbol{T}_\infty$, where $\boldsymbol{\sigma}_\infty$ is the canonical freezing substitution. Then since ϕ is satisfiable, $\boldsymbol{\sigma}_\infty \boldsymbol{T}_\infty$ will have open branches $(B_k)_{k \geq 0}$ (the number need not be finite). It follows from the format of the tableau expansion rules that every open branch will develop one binding.

A binding $\theta \neq []$ occurs non-protected in a formula of the form $\theta; \psi$. Check that the tableau expansion rules on formulas of the forms $(\!(\psi)\!)$ or $\neg(\psi)$ never yield (nontrivial) non-protected bindings. Check that each application of an α, β, γ or δ rule to a formula with a non-protected binding extends a branch with exactly one non-protected binding. It follows that every tableau branch B_k has a highest node where a formula of the form θ appears. This θ can be thought of as the result of pulling the initial binding $[]$ through the initial formula ϕ. For every such B_k and θ there is a finite $\boldsymbol{T}_n$ with a branch $B_{k'}$ that already contains (a generalization of) θ.

It can be proved by induction on the length of $B_{k'}$ that ${}_s[\![\phi]\!]^{\mathcal{M}}_{s_\theta}$, for $\mathcal{M}$ the canonical model and s the canonical valuation for that branch. ■

Note that the computation theorem gives no recipe for generating all correct bindings for a given ϕ. Specifying appropriate computation rules for generating these bindings for specific sets of DFOL formulas remains a topic for future research.

Variation: Using the Calculus with a Fixed Model Computing with respect to a fixed model is but a slight variation on the general scheme. The technique of using tableau rules for model checking is well known. Assume that a model $\mathcal{M} = (D, I)$ is given. Then instead of storing ground predicates $P\theta\bar{t}$ (ground equalities $\theta t_1 \doteq \theta t_2$), we check the model for $\mathcal{M} \models P\theta\bar{t}$ (for $[\![\theta t_1]\!]^{\mathcal{M}} = [\![\theta t_2]\!]^{\mathcal{M}}$), and close the branch if the test fails, continue otherwise. Similarly, instead of storing ground predicates $P\theta\bar{t}$ (ground equalities $\theta t_1 \doteq \theta t_2$) under negation, we check the model for $\mathcal{M} \not\models P\theta\bar{t}$ (for $[\![\theta t_1]\!]^{\mathcal{M}} \neq [\![\theta t_2]\!]^{\mathcal{M}}$), and close the branch if the test fails, continue otherwise.

11 Adding Iteration

Let $\mathcal{L}^*_\Sigma$ be the language that results from extending $\mathcal{L}_\Sigma$ with formulas of the form ϕ^*. The intended relational meaning of ϕ^* is that ϕ gets executed a finite (≥ 0) number of times. This extension makes $\mathcal{L}^*_\Sigma$ into a full-fledged programming language, with its assertion language built in for good measure.

The semantic clause for ϕ^* runs as follows:

$$\begin{array}{lll} {}_s[\![\phi^*]\!]^{\mathcal{M}}_u & \text{iff} & \text{either } s = u \\ & & \text{or } \exists s_1, \ldots, s_n (n \geq 1) \text{ with } {}_s[\![\phi]\!]^{\mathcal{M}}_{s_1}, \ldots, {}_{s_n}[\![\phi]\!]^{\mathcal{M}}_u. \end{array}$$

It is easy to see that it follows from this definition that:

$$ {}_s[\![\phi^*]\!]^{\mathcal{M}}_u \text{ iff either } s = u \text{ or } \exists s_1 \text{ with } {}_s[\![\phi]\!]^{\mathcal{M}}_{s_1} \text{ and } {}_{s_1}[\![\phi^*]\!]^{\mathcal{M}}_u. \qquad (11.1)$$

Note, however, that (11.1) is not equivalent to the definition of ${}_s[\![\phi^*]\!]_u^{\mathcal{M}}$, for (11.1) does not rule out infinite ϕ paths.

Let ϕ^n be given by: $\phi^0 := []$ and $\phi^{n+1} := \phi; \phi^n$. Now ϕ^* is equivalent to 'for some $n \in \mathbb{N} : \phi^n$'.

What we will do in our calculus for DFOL* is take (11.1) as the cue to the star rules. This will allow star computations to loop, which does not pose any problem, given that we extend our notion of closure to 'closure in the limit' (see below).

The calculus for DFOL* has all expansion rules of the DFOL calculus, plus the following α^* and β^* rules.

α^* *expansion rule* Call ψ^* the star formula of the rule.

$$\begin{tikzcd}
\neg(\phi; \psi^*; \chi) \arrow[d, dash] \\
\neg(\phi; \chi) \\[-20pt]
\neg(\phi; \psi; \psi^*; \chi)
\end{tikzcd}$$

β^* *expansion rule* Call ψ^* the star formula of the rule. The β^* rule also has a protected version.

$$\begin{tikzcd}
 & \phi; \psi^*; \chi \arrow[dl, dash] \arrow[dr, dash] & \\
\phi; \chi & & \phi; \psi; \psi^*; \chi
\end{tikzcd}$$

To see that the α^* rule is sound, assume that s universally satisfies $\neg(\phi; \psi^*; \chi)$ in $\mathcal{M} = (D, I)$. By (11.1), this means that there is at least one $h : \boldsymbol{X} \to D$ for which there is no u with ${}_{s\cup h}[\![\phi; \chi]\!]_u^{\mathcal{M}}$ and no u with ${}_{s\cup h}[\![\phi; \psi; \psi^*; \chi]\!]_u^{\mathcal{M}}$. Thus, s universally satisfies $\neg(\phi; \chi)$ and $\neg(\phi; \psi; \psi^*; \chi)$ in $\mathcal{M}$.

For the β^* rule, assume that s universally satisfies $\phi; \psi^*; \chi$ in $\mathcal{M}$. Then for every $h : \boldsymbol{X} \to D$ there are u, u' with ${}_{s\cup h}[\![\phi]\!]_u^{\mathcal{M}}$ and ${}_u[\![\psi^*; \chi]\!]_{u'}^{\mathcal{M}}$. Then, by (11.1), either ${}_u[\![\chi]\!]_{u'}^{\mathcal{M}}$ or there is a u_1 with ${}_u[\![\psi]\!]_{u_1}^{\mathcal{M}}$ and ${}_{u_1}[\![\phi_1^*; \chi]\!]_{u'}^{\mathcal{M}}$. Thus, s universally satisfies either $\phi; \chi$ or $\phi; \psi; \psi^*; \chi$ in $\mathcal{M}$.

Closure in the Limit To deal with the inflationary nature of the α^* and β^* rules (the star formula of the rule reappears at a leaf node), we need a modification of our notion of tableau closure. We allow closure in the limit, as follows.

Definition 11.1
An infinite tableau branch closes in the limit if it contains an infinite star development, i.e., an infinite number of α^* or β^* applications to the same star formula.

Example of Closure in the Limit We will give an example of an infinite star development. Consider formula (11.2):

$$\neg\exists w \neg(\exists v; v = 0; (v \neq w; [v + 1/v])^*; v = w). \tag{11.2}$$

What (11.2) says is that there is no object w that cannot be reached in a finite number of steps from $v = 0$, or in other words that the successor relation $v \mapsto v+1$, considered as a graph, is well-founded. This is the Peano induction axiom: it characterizes the natural numbers up to isomorphism. What it says is that any set A that contains 0 and is closed under successor contains all the natural numbers. The fact that Peano induction is expressible as an $\mathcal{L}^*_\Sigma$ formula is evidence that $\mathcal{L}^*_\Sigma$ has greater expressive power than FOL. In FOL no single formula can express Peano induction: no formula can distinguish the standard model $(\mathbb{N}, s)$ from the non-standard models. In a non-standard model of the natural numbers it may take an infinite number of s-steps to get from one natural number n to a larger number m.

The expressive power of $\mathcal{L}_\Sigma{}^*$ is the same as that of quantified dynamic logic ([27, 17]). Arithmetical truth is undecidable, so there can be no finitary refutation system for $\mathcal{L}^*_\Sigma$. The finitary tableau system for $\mathcal{L}_\Sigma$ is evidence for the fact that DFOL validity is recursively enumerable: all non-validities are detected by a finite tableau refutation. This property is lost in the case of $\mathcal{L}^*_\Sigma$: the language is just too expressive to admit of finitary tableau refutations.

Therefore, some tableau refutations must be infinitary, and the tableau development for the negation of (11.2) is a case in point. Let us see what happens if we attempt to refute the negation of (11.2). A successful refutation will identify the natural numbers up to isomorphism. See Figure 10. This is indeed a successful refutation, for the tree closes in the limit. But the refutation tree is infinite: it takes an infinite amount of time to do all the checks.

Theorem 11.2 (Soundness Theorem for $\mathcal{L}^*_\Sigma$)
The calculus for DFOL* is sound:

For all $\phi, \psi \in \mathcal{L}^*_\Sigma$: if the tableau for $\phi; \neg(\psi)$ closes then $\phi \models \psi$.

The modified tableau method does not always give finite refutations. Still, it is a very useful reasoning tool, more powerful than Hoare reasoning, and more practical than the infinitary calculus for quantified dynamic logic developed in [16, 17]. Dynamic logic itself has been put to practical use, e.g. in KIV, a system for interactive software verification [28]. It is our hope that the present calculus can be used to further automate the software verification process.

Precondition/postcondition Reasoning For a further example of reasoning with the calculus, consider formula (11.3). This gives an $\mathcal{L}^*_\Sigma$ version of Euclid's GCD algorithm.

$$(x \neq y; (x > y; [x - y/x] \cup y > x; [y - x/y]))^*; x \doteq y. \tag{11.3}$$

To do automated precondition-postcondition reasoning on this, we must find a trivial correctness statement. Even if we don't know what $\text{gcd}(x, y)$ is, we know that its value should not change during the program. So putting $\text{gcd}(x, y)$ equal to some arbitrary value and see what happens would seem to be a good start. We will use the correctness statement $z \doteq \text{gcd}(x, y)$. The statement that the result gets computed in x can then take the form $z \doteq x$. The program with these trivial correctness statements included becomes:

$$\begin{aligned}&z \doteq \text{gcd}(x, y);\\&(x \neq y; (x > y; [x - y/x]; z \doteq \text{gcd}(x, y) \cup y > x; [y - x/y]; z \doteq \text{gcd}(x, y)))^*;\\&x \doteq y; z \doteq x.\end{aligned} \tag{11.4}$$

FIG. 10. 'Infinite Proof' of the Peano Induction Axiom.

$$
\begin{array}{c}
\exists w \neg(\exists v; v \doteq 0; (v \neq w; [v+1/v])^*; v \doteq w) \\
| \\
[w_1/w]\neg(\exists v; v \doteq 0; (v \neq w; [v+1/v])^*; v \doteq w) \\
| \\
\neg([w_1/w, 0/v]; (v \neq w; [v+1/v])^*; v \doteq w) \\
| \\
\neg([w_1/w, 0/v]; v \doteq w) \\
\neg([w_1/w, 0/v]; v \neq w; [v+1/v]; (v \neq w; [v+1/v])^*; v \doteq w) \\
| \\
0 \neq w_1 \\
| \\
\neg([w_1/w, 1/v]; (v \neq w; [v+1/v])^*; v \doteq w) \\
| \\
\neg([w_1/w, 1/v]; v \doteq w) \\
\neg([w_1/w, 1/v]; v \neq w; [v+1/v]; (v \neq w; [v+1/v])^*; v \doteq w) \\
| \\
1 \neq w_1 \\
| \\
\neg([w_1/w, 2/v]; (v \neq w; [v+1/v])^*; v \doteq w) \\
| \\
\neg([w_1/w, 2/v]; v \doteq w) \\
\neg([w_1/w, 2/v]; v \neq w; [v+1/v]; (v \neq w; [v+1/v])^*; v \doteq w) \\
| \\
2 \neq w_1 \\
| \\
\neg([w_1/w, 3/v]; (v \neq w; [v+1/v])^*; v \doteq w) \\
| \\
\neg([w_1/w, 3/v]; v \doteq w) \\
\neg([w_1/w, 3/v]; v \neq w; [v+1/v]; (v \neq w; [v+1/v])^*; v \doteq w) \\
| \\
3 \neq w_1 \\
| \\
\neg([w_1/w, 4/v]; (v \neq w; [v+1/v])^*; v \doteq w) \\
| \\
\neg([w_1/w, 4/v]; v \doteq w) \\
\neg([w_1/w, 4/v]; v \neq w; [v+1/v]; (v \neq w; [v+1/v])^*; v \doteq w) \\
| \\
4 \neq w_1 \\
| \\
\neg([w_1/w, 5/v]; (v \neq w; [v+1/v])^*; v \doteq w) \\
| \\
\vdots \\
\times
\end{array}
$$

We can now put the calculus to work. Abbreviating

$$(x \neq y; (x > y; [x - y/x]; z \doteq \mathrm{gcd}(x, y) \cup y > x; [y - x/y]; z \doteq \mathrm{gcd}(x, y)))^*$$

as A^*, we get:

$$\begin{tikzcd}[column sep=small]
 & z \doteq \mathrm{gcd}(x,y); A^*; x \doteq y; z \doteq x \arrow[dl, dash] \arrow[dr, dash] & & \\
{[\mathrm{gcd}(x,y)/z]; x \doteq y; z \doteq x} \arrow[d, dash] & & {[\mathrm{gcd}(x,y)/z]; A; A^*; x \doteq y; z \doteq x} \arrow[dl, dash] \arrow[dr, dash] & \\
x \doteq y, \mathrm{gcd}(x,y) \doteq x & \begin{array}{c} x > y \\ \mathrm{gcd}(x,y) \doteq \mathrm{gcd}(x-y,y) \\ {[\mathrm{gcd}(x,y)/z, x-y/x]; A^*;} \\ x \doteq y; z \doteq x \end{array} & & \begin{array}{c} y > x \\ \mathrm{gcd}(x,y) \doteq \mathrm{gcd}(x,y-x) \\ {[\mathrm{gcd}(x,y)/z, y-x/y]; A^*;} \\ x \doteq y; z \doteq x \end{array}
\end{tikzcd}$$

The second split is caused by an application of the rule for $\cup$. By the soundness of the calculus any model satisfying the annotated program (11.4) will satisfy one of the branches. This shows that if the program succeeds (computes an answer), the following disjunction will be true:

$$\begin{array}{ll} & (x \doteq y \wedge \mathrm{gcd}(x,y) \doteq x) \\ \vee & (x > y \wedge \mathrm{gcd}(x,y) \doteq \mathrm{gcd}(x-y,y) \wedge \phi) \\ \vee & (y > x \wedge \mathrm{gcd}(x,y) \doteq \mathrm{gcd}(x,y-x) \wedge \psi), \end{array} \tag{11.5}$$

where ϕ and ψ abbreviate, respectively, $[\mathrm{gcd}(x,y)/z, x - y/x]; A^*; x \doteq y; z \doteq x$ and $[\mathrm{gcd}(x,y)/z, y - x/y]; A^*; x \doteq y; z \doteq x$. From this it follows that the following weaker disjunction is also true:

$$\begin{array}{ll} & (x \doteq y \wedge \mathrm{gcd}(x,y) \doteq x) \\ \vee & (x > y \wedge \mathrm{gcd}(x,y) \doteq \mathrm{gcd}(x-y,y)) \\ \vee & (y > x \wedge \mathrm{gcd}(x,y) \doteq \mathrm{gcd}(x,y-x)) \end{array} \tag{11.6}$$

Note that (11.6) looks remarkably like a functional program for GCD.

12 Completeness for DFOL*

The method of trace sets for proving completeness from Section 10 still applies. Trace sets for DFOL* will have to satisfy the obvious extra conditions. In order to preserve the correspondence between trace sets and open tableau branches, we must adapt the definition of a fair computation rule. A computation rule F for $\mathcal{L}^*{}_\Sigma$ is fair if it is fair for $\mathcal{L}_\Sigma$, and in addition, the following holds for all branches B in $\boldsymbol{T}_\infty$:

- All formulas of type α^*, β^* occurring on B or in Φ were used to expand B.

We can again prove a trace lemma for DFOL*, in the same manner as before: Again, open branches in the supremum of a fair tableau sequence will correspond to trace sets, and we can satisfy these trace sets in canonical models. The definition of trace sets is extended as follows:

Definition 12.1
A set Ψ of $\mathcal{L}^*_{\Sigma^*}$ formulas is a **$*$-trace set** if the following hold:

- Ψ is a trace set,
- If $\beta^* \in \Psi$ then at least one $\beta^*_i \in \Psi$.
- If $\phi; \psi^*; \chi \in \Psi$, then there is some $n \geq 0$ with $\phi; \psi^m; \chi \notin \Psi$ for all $m > n$. Similarly for $(\!(\phi; \psi^*; \chi)\!)$.
- For all ϕ, ψ, χ it holds that $\neg(\phi; \psi^*; \chi) \notin \Psi$.

Note that the final two requirements are met thanks to our stipulation about closure in the limit. In the same manner as before, we get:

Theorem 12.2 (Completeness for $\mathcal{L}^*$)
For all $\phi, \psi \in \mathcal{L}^*$: if $\phi \models \psi$ then the tableau for $\phi; \neg(\psi)$ closes.

So we have a complete logic for DFOL*, but of course it comes at a price: we may occasionally get in a refutation loop. However, as our tableau construction examples illustrate, this does hardly affect the usefulness of the calculus.

13 Related Work

Comparison with tableau reasoning for (fragments of) FOL The present calculus for DFOL can be viewed as a more dynamic version of tableau style reasoning for FOL and for modal fragments of FOL. Instead of just checking for valid consequence and constructing counterexamples from open tableau branches, our open tableau branches yield computed answer bindings as an extra. The connection with tableau reasoning for FOL is also evident in the proof method of our completeness theorems. Our calculus can be used for FOL reasoning via the following translation of FOL into DFOL:

$$\begin{array}{rcl}
(P\bar{t})^\bullet & := & P\bar{t} \\
(\neg\phi)^\bullet & := & \neg\phi^\bullet \\
(\phi \wedge \psi)^\bullet & := & \phi^\bullet; \psi^\bullet \\
(\phi \vee \psi)^\bullet & := & \phi^\bullet \cup \psi^\bullet \\
(\exists x\phi)^\bullet & := & (\!(\exists x; \phi^\bullet)\!) \\
(\forall x\phi)^\bullet & := & \neg(\exists x; \neg\phi^\bullet)
\end{array}$$

It is easy to check that for every FOL formula ϕ it holds that $\phi^\bullet = \phi^{\bullet\boxdot}$, i.e., all FOL translations are DFOL tests. Moreover, the translation is adequate in the sense that for every FOL formula ϕ over signature Σ, every Σ-model $\mathcal{M}$, every valuation s for $\mathcal{M}$ it holds that $\mathcal{M} \models_s \phi$ iff ${}_s[\![\phi^\bullet]\!]^{\mathcal{M}}_s$.

Connection with Logic Programming The close connection between tableau reasoning for DFOL and Logic Programming can be seen by developing a DFOL tableau for the following formula set:

$$\forall x A([], x, x), \forall x \forall y \forall z \forall i (A(x, y, z) \rightarrow A([i|x], y, [i|z])), \neg\exists x A([a|[b|[]], [c|[]], x).$$

This will give a tableau for the append relation, with a MGU substitution $\{\boldsymbol{x} \mapsto [a|[b|c|[]]]\}$ that closes the tableau, where $\boldsymbol{x}$ is the universal tableau variable used in the application of the γ rule to $\neg\exists x A([a|[b|[]], [c|[]], x)$. The example may serve as a hint to the unifying perspective on logic programming and imperative programming provided by tableau reasoning for DFOL. We hope to elaborate this theme in future work.

Comparison with other Calculi for DFOL and for DRT The calculus developed in [14] uses swap rules for moving quantifiers to the front of formulas. The key idea of the present calculus is entirely different: encode dynamic binding in explicit bindings and protect outside environments from dynamic side effects by means of block operations. In a sense, the present calculus offers a full account of the phenomenon of local variable use in DFOL.

Kohlhase [24] gives a tableau calculus for DRT (Discourse Representation Theory, see [23]) that has essentially the same scope as the [14] calculus for DPL: the version of DRT disjunction that is treated is externally static, and the DRT analogue of $\cup$ is not treated.

The Kohlhase calculus follows an old DRT tradition in relying on an implicit translation to standard FOL: see [29] for an earlier example of this. Kohlhase motivates his calculus with the need for (minimal) model generation in dynamic NL semantics. In order to make his calculus generate minimal models, he replaces the rule for existential quantification by a 'scratchpaper' version (well-known from textbook treatments of tableau reasoning; see [22] for further background, and for discussion of non-monotonic consequence based on minimal models generated with this rule): first try out if you can avoid closure with a term already available at the node. If all these attempts result in closure, it does not follow from this that the information at the node is inconsistent, for it may just be that we have 'overburdened' the available terms with demands. So in this case, and only in this case, introduce a new individual.

This 'exhaustion of existing terms' approach has the virtue that it generates 'small' models when they exist, whereas the more general procedure 'always introduce a fresh variable and postpone instantiation' may generate infinite models where finite models exist. Note, however, that the strategy only makes sense for a signature without function symbols, and for a tableau calculus without free tableau variables.

Kohlhase discusses applications in NL processing, where it often makes sense to construct a minimal model for a text, and where the assumption of minimality can be used to facilitate issues of anaphora resolution and presupposition handling.

Comparison with Apt and Bezem's Executable FOL Apt and Bezem present what can be viewed as an exciting new mix of tableau style reasoning and model checking for FOL. Our treatment of equality uses a generalization of a stratagem from their [3]: in the context of a partial variable map θ, they call $v \doteq t$ a θ *assignment* if $v \notin dom(\theta)$, and all variables occurring in t are in $dom(\theta)$. We generalize this on two counts:

- Because our computation results are bindings (term maps) rather than maps to objects in the domain of some model, we allow computation of non-ground terms as values.
- Because our bindings are total, in our calculus execution of $t_1 \doteq t_2$ atoms never gives rise to an error condition.

It should be noted for the record that the first of these points is addressed in [2]. Apt and Bezem present their work as an underpinning for Alma-0, a language that infuses

Modula style imperative programming with features from logic programming (see [4]). In a similar way, the present calculus provides logical underpinnings for Dynamo, a language for programming with an extension of DFOL. For a detailed comparison of Alma-0 and Dynamo we refer the reader to [13].

Connection with WHILE, GCL It is easy to give an explicit binding semantics for WHILE, the favorite toy language of imperative programming from the textbooks (see e.g., [25]), or for GCL, the non-deterministic variation on this proposed by Dijkstra (see, e.g. [10]). DFOL is in fact quite closely related to these, and it is not hard to see that DFOL* has the same expressive power as GCL. Our tableau calculus for DFOL* can therefore be regarded as an execution engine *cum* reasoning engine for WHILE or GCL.

Connection with PDL, QDL There is also a close connection between DFOL* on one hand and propositional dynamic logic (PDL) and quantified dynamic logic (QDL) on the other. QDL is a language proposed in [27] to analyze imperative programming, and PDL is its propositional version. See [30, 26] for complete axiomatizations of PDL, [17] for an exposition of both PDL and QDL, and for a complete (but infinitary) axiomatization of QDL, [21] for an overview, and [20] for a a study of QDL and various extensions. In PDL/QDL, programs are treated as modalities and assertions about programs are formulas in which the programs occur as modal operators. Thus, if A is a program, $\langle A\rangle\phi$ asserts that A has a successful termination ending in a state satisfying ϕ. As is well-known, this cannot be expressed without further ado in Hoare logic.

The main difference between DFOL* and PDL/QDL is that in DFOL* the distinction between formulas and programs is abolished. Everything is a program, and assertions about programs are test programs that are executed along the way, but with their dynamic effects blocked. To express that A has a successful termination ending in a ϕ state, we can just say $(\!(A;\phi)\!)$. To check whether A has a successful termination ending in a ϕ state, try to refute the statement by constructing a tableau for $\neg(A;\phi)$.

To illustrate the connection with QDL and PDL, consider MIX, the first of the two PDL axioms for $*$:

$$[A^*]\phi \rightarrow \phi \wedge [A][A^*]\phi. \tag{13.1}$$

Writing this with $\langle A\rangle, \neg, \wedge, \vee$, and replacing $\neg\phi$ by ϕ, we get:

$$\neg(\neg\langle A^*\rangle\phi \wedge (\phi \vee \langle A\rangle\langle A^*\rangle\phi)). \tag{13.2}$$

This has the following DFOL* counterpart:

$$\neg(\neg(A^*;\phi);(\phi \cup (A;A^*;\phi))). \tag{13.3}$$

For a refutation proof of (13.3), we leave out the outermost negation.

$$\begin{tikzcd}[column sep=small, row sep=small]
& \neg(A^*;\phi);(\phi\cup(A;A^*;\phi)) \arrow[d, dash] & \\
& \neg(A^*;\phi) & \\
& (\phi\cup(A;A^*;\phi)) \arrow[d, dash] & \\
& \neg\phi & \\
& \neg(A;A^*;\phi) \arrow[dl, dash] \arrow[dr, dash] & \\
\phi & & (A;A^*;\phi) \\
\times & & \times
\end{tikzcd}$$

The tableau closes, so we have proved that (13.3) is a DFOL* theorem (and thus, a DFOL* validity).

We will also derive the validity of the DFOL* counterpart to IND, the other PDL axiom for $*$:

$$(\phi \wedge [A^*](\phi \rightarrow [A]\phi)) \rightarrow [A^*]\phi. \tag{13.4}$$

Equivalently, this can be written with only $\langle A\rangle, \neg, \wedge, \vee$, as follows:

$$\neg(\phi \wedge \neg\langle A^*\rangle(\phi \wedge \langle A\rangle\neg\phi) \wedge \langle A^*\rangle\neg\phi). \tag{13.5}$$

The DFOL* counterpart of (13.5) is:

$$\neg(\phi; \neg(A^*;\phi;A;\neg\phi);A^*;\neg\phi). \tag{13.6}$$

We will give a refutation proof of (13.6) in two stages. First, we show that (13.7) can be refuted for any $n \geq 0$, and next, we use this for the proof of (13.6).

$$\phi;\neg(A^*;\phi;A;\neg\phi);A^n;\neg\phi. \tag{13.7}$$

Here is the case of (13.7) with $n = 0$:

$$\begin{tikzcd}[row sep=small]
\phi;\neg(A^*;\phi;A;\neg\phi);\neg\phi \arrow[d, dash] \\
\phi \\
\neg(A^*;\phi;A;\neg\phi) \\
\neg\phi \\
\times
\end{tikzcd}$$

Bearing in mind that A is a dynamic action and ϕ is a test, we can apply the rule of Negation Splitting to formulas of the form $\neg(A^n;\phi;A;\neg\phi)$, as follows:

$$\begin{tikzcd}[column sep=small, row sep=small]
& \neg(A^n;\phi;A;\neg\phi) \arrow[dl, dash] \arrow[dr, dash] & \\
((A^n;\neg\phi)) & & \neg(A^{n+1};\neg\phi)
\end{tikzcd}$$

Note that $\neg(A^n;\phi;A;\neg\phi)$ can be derived from $\neg(A^*;\phi;A;\neg\phi)$ by n applications of the α^* rule. Using this, we get the following refutation tableau for the case of (13.7) with $n = k + 1$:

$$\begin{tikzcd}[column sep=small]
& \phi; \neg(A^*; \phi; A; \neg\phi); A^{k+1}; \neg\phi \arrow[d, dash] & \\
& \begin{array}{c} \phi \\ \neg(A^*; \phi; A; \neg\phi) \\ A^{k+1}; \neg\phi \end{array} \arrow[d, dash] & \\
& \vdots \arrow[d, dash] & \\
& \neg(A^k; \phi; A; \neg\phi) \arrow[dl, dash] \arrow[dr, dash] & \\
\begin{array}{c} ((A^k; \neg\phi)) \\ \times \end{array} & & \begin{array}{c} \neg(A^{k+1}; \neg\phi) \\ \times \end{array}
\end{tikzcd}$$

The left-hand branch closes because of the refutation of $\phi; \neg(A^*; \phi; A; \neg\phi); A^k; \neg\phi$, which is given by the induction hypothesis.

Next, use these refutations of $\neg\phi$, $A; \neg\phi$, $A^2; \neg\phi$, $\dots$, to prove (13.6) by means of a refutation in the limit, as follows:

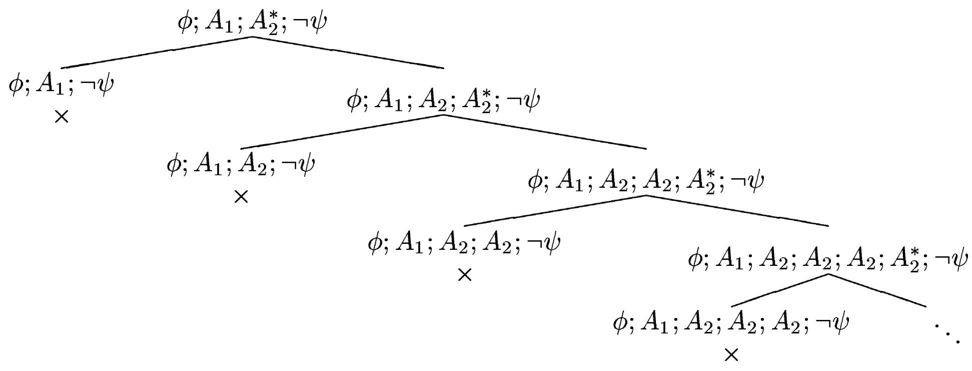
$$\begin{tikzcd}[column sep=small]
& \phi; \neg(A^*; \phi; A; \neg\phi); A^*; \neg\phi \arrow[d, dash] & & & \\
& \begin{array}{c} \phi \\ \neg(A^*; \phi; A; \neg\phi) \\ A^*; \neg\phi \end{array} \arrow[dl, dash] \arrow[dr, dash] & & & \\
\begin{array}{c} \neg\phi \\ \times \end{array} & & A; A^*; \neg\phi \arrow[dl, dash] \arrow[dr, dash] & & \\
& \begin{array}{c} A; \neg\phi \\ \times \end{array} & & A^2; A^*; \neg\phi \arrow[dl, dash] \arrow[dr, dash] & \\
& & \begin{array}{c} A^2; \neg\phi \\ \times \end{array} & & A^3; A^*; \neg\phi \arrow[dl, dash] \arrow[dr, dash] & \\
& & & A^3; \neg\phi & & \ddots
\end{tikzcd}$$

This closed tableau establishes (13.6) as a DFOL* theorem. That closure in the limit is needed to establish the DFOL* induction principle is not surprising. The DFOL * rules express that * computes a fix-point, while the fact that this fix-point is a *least* fix-point is captured by the stipulation about closure in the limit. The induction principle (13.6) hinges on the fact that * computes a least fix-point.

Goldblatt [16, 17] develops an infinitary proof system for QDL with the following key rule of inference:

If $\phi \rightarrow [A_1; A_2^n]\psi$ is a theorem for every $n \in \mathbb{N}$, then $\phi \rightarrow [A_1; A_2^*]\psi$ is a theorem. (13.8)

To see how this is related to the present calculus, assume that one attempts to refute $\phi \rightarrow [A_1; A_2^*]\psi$, or rather, its DFOL* counterpart $\neg(\phi; A_1; A_2^*; \neg\psi)$, on the assumption that for any $n \in \mathbb{N}$ there exists a refutation of $\phi; A_1; A_2^n; \neg\psi$.


$\phi; A_1; A_2^*; \neg\psi$
$\phi; A_1; \neg\psi$
$\times$
$\phi; A_1; A_2; A_2^*; \neg\psi$
$\phi; A_1; A_2; \neg\psi$
$\times$
$\phi; A_1; A_2; A_2; A_2^*; \neg\psi$
$\phi; A_1; A_2; A_2; \neg\psi$
$\times$
$\phi; A_1; A_2; A_2; A_2; A_2^*; \neg\psi$
$\phi; A_1; A_2; A_2; A_2; \neg\psi$
$\times$
$\ddots$

We can close off the $\phi; A_1; A_2^n; \neg\psi$ branches by the assumption that there exist refutations for these, for every $n \in \mathbb{N}$. The whole tableau gives an infinite β^* development, and the infinite branch closes in the limit, so the tableau closes, thus establishing that in the DFOL* calculus validity of $\neg(\phi; A_1; A_2^*; \neg\psi)$ follows from the fact that $\neg(\phi; A_1; A_2^n; \neg\psi)$ is valid for every $n \in \mathbb{N}$.

14 Conclusion

Starting out from an analysis of binding in dynamic FOL, we have given a tableau calculus for reasoning with DFOL. The format for the calculus and the role of explicit bindings for computing answers to queries were motivated by our search for logical underpinnings for programming with (extensions of) DFOL. The DFOL tableau calculus presented here constitutes the theoretical basis for *Dynamo*, a toy programming language based on DFOL. The versions of *Dynamo* implemented so far implement tableau reasoning for DFOL with respect to a fixed model: see [13].

To find the answer to a query, given a formula ϕ considered as *Dynamo* program data, *Dynamo* essentially puts the tableau calculus to work on a formula ϕ, all the while checking predicates with respect to the fixed model of the natural numbers, and storing values for variables from the inspection of equality statements. If the tableau closes, this means that ϕ is inconsistent (with the information obtained from testing on the natural numbers), and *Dynamo* reports 'false'. If the tableau remains open, *Dynamo* reports that ϕ is consistent (again with the information obtained from inspecting predicates on the natural numbers), and lists the computed bindings for the output variables at the end of the open branches. But the *Dynamo* engine also works for general tableau reasoning, and for general queries. The literals collected along the open branches together with the explicit bindings at the trail ends constitute the computed answers.

Dynamo can be viewed as a combined engine for program execution and reasoning. We are currently working on an new implementation of *Dynamo* that takes the insights reported above into account. The advantages of the combination of execution and reasoning embodied in *Dynamo* should be evident from our examples of strongest postcondition generation in Section 9. To our knowledge, this use of dynamic first order logic for analyzing imperative programming by means of calculating trace sets is new. We claim that our calculus opens the road to a more intuitive way of reasoning about imperative programs, and we hope to develop automated reasoning tools for program analysis based on it.

Finally, since natural language semantics is a key application area of dynamic variations on first order logic, we expect that both the calculus itself and its implementation in the form of an improved execution mechanism for *Dynamo* also have a role to play in a truly computational semantics for natural language.

Acknowledgments

The research for this paper was sponsored by *Spinoza Logic In Action*. Thanks to Johan van Benthem, Balder ten Cate, Anne Kaldewaij, Fairouz Kamareddine, Michael Kohlhase, Maarten Marx, Joachim Niehren, Kees Vermeulen, Albert Visser and Joe Wells for stimulating discussion and helpful criticism. Two anonymous reviewers of this journal made suggestions that prompted a complete overhaul of the presentation. Proposition (3.4) was triggered by a question from Krzysztof Apt.

References

[1] K.R. Apt. *From Logic Programming to Prolog.* Prentice Hall, 1997.

[2] K.R. Apt. A denotational semantics for first-order logic. In *Proc. of the Computational Logic Conference (CL2000)*, Notes in Artificial Intelligence 1861, pages 53–69. Springer, 2000.

[3] K.R. Apt and M. Bezem. Formulas as programs. In K.R. Apt, V. Marek, M. Truszczyski, and D.S. Warren, editors, *The Logic Programming Paradigm: a 25 Years Perspective*, pages 75–107. Springer Verlag, 1999. Paper available as `http://xxx.lanl.gov/abs/cs.LO/9811017`.

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

[5] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[6] J. van Benthem. Partiality and nonmonotonicity in classical logic. *Logique et Analyse*, 29, 1986.

[7] J. van Benthem. *Exploring Logical Dynamics.* CSLI & Folli, 1996.

[8] J. van Benthem and J. van Eijck. The dynamics of interpretation. *Journal of Semantics*, 1(1):3–20, 1982.

[9] M. D'Agostino, D.M. Gabbay, R. Hähnle, and J. Posegga, editors. *Handbook of Tableau Methods.* Kluwer, Dordrecht, 1999.

[10] E.W Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics.* Texts and Monographs in Computer Science. Springer-Verlag, 1990.

[11] H.C. Doets. *From Logic to Logic Programming.* MIT Press, Cambridge, Massachusetts, 1994.

[12] G. Dowek, T. Hardin, and C. Kirchner. Theorem proving modulo. Technical Report 3400, INRIA Rocquencourt, April 1998.

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

[14] J. van Eijck. Axiomatising dynamic logics for anaphora. *Journal of Language and Computation*, 1:103–126, 1999.

[15] M. Fitting. *First-order Logic and Automated Theorem Proving; Second Edition.* Springer Verlag, Berlin, 1996.

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

[17] R. Goldblatt. *Logics of Time and Computation, Second Edition, Revised and Expanded*, volume 7 of *CSLI Lecture Notes.* CSLI, Stanford, 1992 (first edition 1987). Distributed by University of Chicago Press.

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

[19] R. Hähnle. Tableaux and related methods. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning.* Elsevier Science Publishers, to appear, 2001.

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

[21] D. Harel. Dynamic logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, pages 497–604. Reidel, Dordrecht, 1984. Volume II.

[22] J. Hintikka. Model minimization — an alternative to circumscription. *Journal of Automated Reasoning*, 4:1–13, 1988.

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

[24] M. Kohlhase. Model generation for Discoure Representation Theory. In *ECAI Proceedings*, 2000. Available from `http://www.ags.uni-sb.de/~kohlhase/`.

[25] H.R. Nielson and F. Nielson. *Semantics with Applications.* John Wiley and Sons, 1992.

[26] R. Parikh. The completeness of propositional dynamic logic. In *Mathematical Foundations of Computer Science 1978*, pages 403–415. Springer, 1978.

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

[28] W. Reif. The KIV-approach to software verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software*, Springer LNCS 1009, pages 339–368, 1995.

[29] C. Sedogbo and M. Eytan. A tableau calculus for DRT. *Logique et Analyse*, 31:379–402, 1988.

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

[31] R. Smullyan. *First-order logic.* Springer, Berlin, 1968.

[32] Y. Venema. A modal logic of quantification and substitution. In L. Czirmaz, D.M. Gabbay, and M. de Rijke, editors, *Logic Colloquium '92*, Studies in Logic, Language and Computation, pages 293–309. CSLI and FOLLI, 1995.

[33] A. Visser. Contexts in dynamic predicate logic. *Journal of Logic, Language and Information*, 7(1):21–52, 1998.

[34] A. Visser. A note on substitution in dynamic semantics. Unpublished draft, Utrecht University, 2000.

Received Sept 8, 2000. Revised: December 9, 2000