Placeholder calculus for first-order logic

Placeholder Calculus for First-Order Logic

Michael Franssen (m.franssen@tue.nl)

Eindhoven University of Technology,
Dept. of Mathematics and Computer Science,
Den Dolech 2, 5612 AZ Eindhoven, The Netherlands

Abstract. In this paper we present an extension of first-order predicate logic with placeholders. These placeholders allow the construction of proofs for incomplete theorems. These theorems can be completed during the proof construction process. By using special definitions of substitutions and replacements, we obtain an unexpectedly simple calculus. Furthermore, we avoid the need of additional rules for explicit substitutions to deal with postponed substitutions in placeholders, since the definitions of substitution and replacement deal with them directly.

1 Introduction

When using an interactive system to prove logical theorems, one usually starts by formulating a goal. However, often it is not entirely clear what the goal should be. Sometimes, the reason for constructing a proof is to find out what the premisses for a theorem exactly are. In such cases, the goal has to be reformulated several times, before the theorem is actually provable. Most systems require a full restart of the proof if the goal is changed. The reason is, that these systems are unable to deal with a goal that is not complete.

Another situation in which the goal is not completely clear from the beginning is program derivation. In the Eindhoven style of programming [8, 2], programs are derived from their specifications. Sometimes, an assignment is inserted in which the expression part is not yet filled in. The expression part of the assignment is then chosen during the construction of the correctness proof at a point where it becomes absolutely clear which expression suffices. This way, the program is truly derived from its specification.

Cocktail [3] is a tool that partially supports the Eindhoven style of programming. Cocktail's logic [9] is based on type theory, even where the automated theorem prover is concerned [4]. However, Cocktail cannot deal with incomplete goals either. In the current development of Cocktail we no longer attempt to encode everything into the type system, but we still restrict ourselves to first-order predicate logic with equalities, which is common since the beginning of formal program development [1, 7]. This way, we hope to allow for easy connection of external theorem provers to the program derivation engine.

In [6], Geuvers and Jojgov show how pure type systems can be extended to deal with open terms and open proofs. In their paper, the focus is on the representation of incomplete λ-terms that represent goals and proofs. They use this to provide a formal basis for tools dealing with incomplete proofs. In the pure type systems they consider, proofs can also contain part of the goal, since they allow for higher order logic. The context of their λ-calculus provides enough information to avoid problems when filling in meta-variables.

Gabbay and Mathijssen propose a one-and-a-halfth-order logic [5]. This logic is capable to handle the problems considered in this paper, but is more complicated. It maintains a set of constraints during the derivation of correct judgements. The axioms of their derivation system then contains side-conditions that are decidable because of this set of constraints. Also, their approach is specifically tailored to extend first-order logic.

In this paper, we will show how a placeholder calculus can be constructed without the need for a λ-calculus. Instead, we will put a simple constraint on the terms that may be filled in for a placeholder. This constraint will not change during the construction of a proof. The rules of the derivation system are exactly the same as those for first-order logic, except that placeholders may be instantiated at any time. Instantiation is not an extra derivation rule, but more a feature of the system as a whole. Moreover, our calculus does not contain additional rules to deal with explicit substitutions. An administration of suspended substitutions in placeholders is maintained, but these suspended substitutions are executed directly when a placeholder is (partially) filled in.

Our placeholder calculus allows us to start constructing a proof for a goal that is not yet complete (i.e. it contains placeholders). During the proving process, we choose instantiations for the placeholders based on the needs of the proof. When all placeholders are filled in, we have a proof that could also have been constructed in the original logic, hence soundness is evident.

We start with an example in section 2. Then, after giving some preliminaries in section 3, we give a simple version of our placeholder calculus in section 4. This simple calculus is extended in section 5 to allow for some more advanced applications. Finally, we give a discussion in section 6.

2 Example

Consider a tool to derive correct programs. During the derivation of a simple program to compute factorials, one encounters the following situation:

$$
\begin{array}{l}
|[\textbf{con } N : \mathit{int}\ \{0 \leq N\} \\
\quad \textbf{var } n, r : \mathit{int}; \\
\quad n := 0; \quad r := 1; \{\mathit{inv} : r = n!\} \\
\quad \textbf{while } (n \neq N) \textbf{ do} \\
\quad\quad \{r = n! \wedge n \neq N\} \\
\quad\quad \ldots \\
\quad\quad \{r = n!\} \\
\quad \textbf{od } \{r = n! \wedge n = N\} \\
\quad \{r = N!\} \\
]|
\end{array}
$$

At the position of the dots, the statement $n := n + 1$ must be filled in, along with an assignment to r to maintain the invariant. The assignment to r can be put before or after the increment of n. When working with pen and paper, the dots are therefore replaced by $r := E; n := n + 1$ or $n := n + 1; r := E$. To prove the correctness of these statements, one has to prove $r = n! \wedge n \neq N \Rightarrow (r \mapsto E)((n \mapsto n + 1)(r = n!))$ or $r = n! \wedge n \neq N \Rightarrow (n \mapsto n + 1)((r \mapsto E)(r = n!))$ respectively. These proofs can be written as follows:

$$
\begin{array}{l}
|[r = n! \wedge n \neq N \\
\quad \triangleright\ (r \mapsto E)((n \mapsto n + 1)(r = n!)) \\
\quad \equiv \{\text{Substitution twice}\} \\
\quad\quad E = (n + 1)! \\
\quad \equiv \{\text{Definition factorial}\} \\
\quad\quad E = (n + 1) * n! \\
\quad \equiv \{\text{Context: } r = n!\} \\
\quad\quad E = (n + 1) * r \\
\quad \equiv \{\bullet \text{ Choose: } E = (n + 1) * r\} \\
\quad\quad \mathit{True} \\
]|
\end{array}
\qquad
\begin{array}{l}
|[r = n! \wedge n \neq N \\
\quad \triangleright\ (n \mapsto n + 1)((r \mapsto E)(r = n!)) \\
\quad \equiv \{\text{Substitution twice}\} \\
\quad\quad (n \mapsto n + 1)E = (n + 1)! \\
\quad \equiv \{\text{Definition factorial}\} \\
\quad\quad (n \mapsto n + 1)E = (n + 1) * n! \\
\quad \equiv \{\text{Context: } r = n!\} \\
\quad\quad (n \mapsto n + 1)E = (n + 1) * r \\
\quad \equiv \{\bullet \text{ Choose: } E = n * r\} \\
\quad\quad (n \mapsto n + 1)(n * r) = (n + 1) * r \\
\quad \equiv \{\text{Substitution}\} \\
\quad\quad \mathit{True} \\
]|
\end{array}
$$

By using a placeholder E instead of an expression of the programming language, the actual statement can be computed from the specification, by choosing an appropriate term for E during the construction of the proof. On pen and paper it is obvious that the substitution in $(n \mapsto n + 1)E$ cannot be performed before E has been chosen.

In the Eindhoven teaching curriculum this method of program derivation is taught to students using pen and paper. However, since it is not really a formal

system, students tend to make many mistakes, in thinking that $(n \mapsto n+1)E$ is equal to E, since any possible occurrence of n is not visible in E.

In a formal logic these derivations are usually not possible. Before a proof can be constructed, a goal must be given as a complete term of the logic. That is, a placeholder like E cannot be used, since it is not a part of the formal logic. In this paper, a formal definition for first order logic with a natural deduction system is given, in which placeholders are part of the formalism. Hence, computations like the proofs given above are allowed within this logic.

The method in which the usual formalism for first order logic is extended with placeholders can also be applied to other formalisms, like modal logics, tableaux calculi etc.

3 Preliminaries

The grammar for formulas F of first-order logic is given by

$$\begin{aligned} F &::= \bot \mid F \Rightarrow F \mid (\forall \mathcal{V}.F) \mid \mathcal{P}(T^*) \\ T &::= \mathcal{F}(T^*) \mid \mathcal{V} \end{aligned}$$

Where $\mathcal{V}$ is a set of variables, $\mathcal{F}$ is a set of function symbols and $\mathcal{P}$ is a set of predicate symbols. Function symbols of arity 0 are called constants.

We will not consider $\top$, $\wedge$, $\vee$, $\neg$ and $\exists$ separately. In classical first-order logic, these operators can all be encoded using $\Rightarrow$ and $\bot$ (False).

$FV(P)$ computes the free variables of a (set of) formula(s) P as usual. $BV(P)$ computes all bound variable of a (set of) formula(s) P as usual. A substitution $\theta : \mathcal{V} \to T$ is a mapping from variables to terms.

The domain of substitution θ is $Dom(\theta) = \{v \in \mathcal{V} | \theta(v) \neq v\}$ and its range is $Range(\theta) = \{v \in Dom(\theta) \mid \theta(v)\}$. We consider only substitutions with a finite domain and denote them by $(x_1 \mapsto \theta(x_1), \ldots, x_n \mapsto \theta(x_n))$ where $\{x_1, \ldots, x_n\} = Dom(\theta)$. Substitutions can also be applied to formulas as usual. The only non-trivial case is substitution in a universal quantification, which is defined as $\theta(\forall x.P) = (\forall y.\theta((x \mapsto y)(P)))$, where $y \notin (Dom(\theta) \cup FV(Range(\theta))) \cup (FV(P) \setminus \{x\})$ (Note that y may be x). This is accomplished by using only *fresh* variables for variable renaming, i.e. names that have not been used before and that will not be used in any other way.

A judgement has the form $\Gamma \vdash P$, where Γ is a set of formulas and $P \in F$. A derivation system S is a set of rules of the form

$$\frac{\Gamma_1 \vdash P_1 \ldots \Gamma_n \vdash P_n}{\Gamma \vdash P}$$

Meaning that if $\Gamma_1 \vdash P_1$ till $\Gamma_n \vdash P_n$ are valid judgements, then so is $\Gamma \vdash P$. $\Gamma_i \vdash P_i$ are called premises and $\Gamma \vdash P$ is called the conclusion of the rule. If n is 0, the rule is called an axiom.

A judgement $\Gamma \vdash P$ is derivable in S if it matches the conclusion of a rule and all the premises of this rule are derivable in S. We denote this by $\Gamma \vdash_S P$.

The conclusions of axioms are, by definition, derivable.

A minimal natural deduction-like derivation system for first order logic as defined above is given by the following rules:

$$\Rightarrow\text{-}I \ \frac{\Gamma \cup \{A\} \vdash B}{\Gamma \vdash A \Rightarrow B} \qquad \Rightarrow\text{-}E \ \frac{\Gamma \vdash A \Rightarrow B \ \Gamma \vdash A}{\Gamma \vdash B}$$

$$\forall\text{-}I \ \frac{\Gamma \vdash A}{\Gamma \vdash (\forall x.A)} \text{ if } x \notin FV(\Gamma) \qquad \forall\text{-}E \ \frac{\Gamma \vdash (\forall x.A)}{\Gamma \vdash (x \mapsto t)(A)}$$

$$\text{falsum} \ \frac{\Gamma \vdash \bot}{\Gamma \vdash A} \qquad \text{context} \ \frac{}{\Gamma \cup \{A\} \vdash A}$$

$$\text{classic} \ \frac{\Gamma \vdash (A \Rightarrow \bot) \Rightarrow \bot}{\Gamma \vdash A}$$

In practice, derivation systems are often used backwards. That is, a goal (theorem, lemma, etc) is formulated as a judgement $\Gamma \vdash P$ and then one attempts to construct a derivation tree within the logic to prove the validity of the judgment. This application is called goal directed reasoning.

A drawback of proving theorems this way is that the theorem has to be fully formulated in order to construct a proof. In daily mathematical practice, the goal or context of the theorem is not entirely known in advance. Instead, during the proving of the theorem the user may want to add items in the context or change the exact formulation of the goal.

4 Placeholder calculus

Our proposal is to introduce placeholders that can be used within the context or the goal and that can be filled in at a later stage. A problem arises when a substitution is applied to a placeholder. Since the placeholder has not yet been filled in, the outcome of the substitution cannot be computed. Therefore, we will also introduce a syntactical notation to represent substitution. When a substitution is applied to a placeholder, a syntactical representation of the substitution is placed behind the placeholder. When the placeholder is filled in, this syntactical representation is converted into a substitution and applied to the image of the placeholder immediately.

We introduce placeholders $\mathcal{H}_N$ for every non-terminal N and extend the grammar with the following rules (we use subscript $\mathcal{H}$ to denote the non-terminals of the extended logic):

$$\begin{array}{l} N_{\mathcal{H}} ::= \mathcal{H}_N S_{\mathcal{H}} \text{ for every non-terminal } N \\ S_{\mathcal{H}} ::= [\mathcal{V}^n := {T_{\mathcal{H}}}^n] \end{array}$$

where $S_{\mathcal{H}}$ is a new non-terminal and $n \in I\!N$.

Substitutions remain mappings from variables to terms of T, not $T_{\mathcal{H}}$.

The lifting function $\uparrow$: $S_{\mathcal{H}} \to (\mathcal{V} \to T)$ converts a syntactical representation of a substitution to a real substitution and is defined as:

$$\uparrow ([x_1, \ldots, x_n := t_1, \ldots, t_n]) = (x_1 \mapsto t_1, \ldots, x_n \mapsto t_n)$$

The dropping function $\downarrow$: $(\mathcal{V} \to T) \to S_{\mathcal{H}}$ converts a substitution to its syntactical representation and is defined as:

$$\downarrow ((x_1 \mapsto t_1, \ldots, x_n \mapsto t_n)) = [x_1, \ldots, x_n := t_1, \ldots, t_n]$$

A replacement $\sigma : \mathcal{H}_N \to N_{\mathcal{H}}$ is a mapping from placeholders to terms of the corresponding non-terminal. The domain $Dom(\sigma)$ is defined as $\{h \in \mathcal{H}_N \mid \sigma(h) \neq h\}$. We consider only replacements with a finite domain and denote them by $(h_1 \rightsquigarrow \sigma(h_1), \ldots, h_n \rightsquigarrow \sigma(h_n))$ where $\{h_1, \ldots, h_n\} = Dom(\sigma)$. Like substitutions, replacements can be applied to formulas of the extended grammar, but the definition is slightly different:

$$\begin{array}{l} \sigma(\forall x.P) = (\forall x.\sigma(P)) \\ \sigma(H[v_1, \ldots, v_n := t_1, \ldots, t_n]) = \uparrow ([v_1, \ldots, v_n := t_1, \ldots, t_n])(\sigma(H)) \end{array}$$

The free variables of a placeholder are not defined. This poses a problem in the definition of applying substitutions to formulas of the extended grammar, since the side-condition $y \notin FV(P) \setminus \{x\}$ cannot be decided. However, this can be remedied by using a special subset of $\mathcal{V}$ for choosing new names for bound variables that is never used in replacements. That is, a placeholder is never mapped to a term containing a variable that was used to rename an existing bound variable, but a placeholder can be mapped to a term containing a bound variable from the original goal.

Applying a substitution θ to a placeholder is defined as follows:

$$\theta(H[v_1, \ldots, v_n := t_1, \ldots, t_n]) = H \downarrow (\theta \circ \uparrow ([v_1, \ldots, v_n := t_1, \ldots, t_n]))$$

The derivation rules for theorems in the extended signature will remain the same as the original derivation rules of the formalism. Filling in a placeholder is an operation on a derivation, rather than on a single judgement.

In order to prove that the correctness of the derivation is not affected by replacements, we have to prove for every derivation rule:

$$\frac{\Gamma_1 \vdash A_1, \ldots \Gamma_n \vdash A_n}{\Gamma \vdash A} \text{ implies } \frac{\sigma(\Gamma_1) \vdash \sigma(A_1), \ldots \sigma(\Gamma_n) \vdash \sigma(A_n)}{\sigma(\Gamma) \vdash \sigma(A)}$$

This proof is mostly trivial. For instance, the context rule can be proved directly by application of the replacement: $\sigma(\Gamma \cup \{A\}) \vdash \sigma(A)$ is equal to $\sigma(\Gamma) \cup \{\sigma(A)\} \vdash \sigma(A)$, which holds directly by the context rule itself.

Complications arise when substitutions occur in one of the premises or goals. Consider the rule $\forall - E$: we have to prove

$$\frac{\sigma(\Gamma) \vdash \sigma((\forall x.A))}{\sigma(\Gamma) \vdash \sigma((x \mapsto t)(A))}$$

Following the definition of replacement, the premise is equal to $\sigma(\Gamma) \vdash (\forall x.\sigma(A))$. From this premise, we can easily derive $(x \mapsto t)(\sigma(A))$. Hence, it is sufficient to prove $\sigma((x \mapsto t)(A)) \equiv (x \mapsto t)(\sigma(A))$, which follows from theorem 1.

Theorem 1. *Let A be a term in the extended signature, let θ be a substitution and σ be a replacement. Then $\sigma(\theta(A)) \equiv \theta(\sigma(A))$.*

Proof. By induction on the structure of A. In most cases this is trivial. We consider the cases $A \equiv (\forall y.A')$ and $A \equiv H[v_1, \ldots, v_n := t_1, \ldots, t_n]$:

Case $A \equiv (\forall y.A')$:
$\sigma(\theta(\forall y.A'))$
$\equiv$ { *Substitution using renaming* }
$\sigma(\forall y'.\theta((y \mapsto y')(A')))$
$\equiv$ { *Function composition* }
$\sigma(\forall y'.(\theta \circ (y \mapsto y'))(A'))$
$\equiv$ { *Definition replacement* }
$(\forall y'.\sigma((\theta \circ (y \mapsto y'))(A')))$
$\equiv$ { *By induction hypothesis* }
$(\forall y'.(\theta \circ (y \mapsto y'))(\sigma(A')))$
$\equiv$ { *Substitution backward* }
$\theta(\forall y.\sigma(A'))$
$\equiv$ { *Replacement backward* }
$\theta(\sigma(\forall y.A'))$

Case $A \equiv H[v_1, \ldots, v_n := t_1, \ldots, t_n]$:
$\sigma(\theta(H[v_1, \ldots, v_n := t_1, \ldots, t_n]))$
$\equiv$ { *Definition substitution* }
$\sigma(H \downarrow (\theta \circ \uparrow ([v_1, \ldots, v_n := t_1, \ldots, t_n])))$
$\equiv$ { *Definition replacement* }
$\uparrow (\downarrow (\theta \circ \uparrow ([v_1, \ldots, v_n := t_1, \ldots, t_n])))(\sigma(H))$
$\equiv$ { $\uparrow (\downarrow (\sigma')) = \sigma'$ }
$\theta \circ \uparrow ([v_1, \ldots, v_n := t_1, \ldots, t_n])(\sigma(H))$
$\equiv$ { *Function composition* }
$\theta(\uparrow ([v_1, \ldots, v_n := t_1, \ldots, t_n])(\sigma(H)))$
$\equiv$ { *Definition replacement* }
$\theta(\sigma(H[v_1, \ldots, v_n := t_1, \ldots, t_n]))$

5 Extending the calculus

Using placeholders and the derivation system above is possible, but limited. For instance, in our example we compute a proof obligation by $(r \mapsto E)((n \mapsto n+1)(r = n!))$, but this is not allowed in the calculus we proposed so far, since $(r \mapsto E)$ is not a mapping $\mathcal{V} \to T$, but a mapping $\mathcal{V} \to T_{\mathcal{H}}$.

The reason to limit the substitutions to $\mathcal{V} \to T$ is the side-condition of substitution in universal quantifications. Consider the following example: $(E \rightsquigarrow x)((y \mapsto E)(\forall x.P(y)))$ equals $(E \rightsquigarrow x)(\forall x.P(E))$ equals $(\forall x.P(x))$, which is clearly wrong. Replacements may contain references to bound variables, but due to the substitution, the placeholder was put in a place where it originally did not occur. $(y \mapsto E)$ puts E in place of y, hence the only valid instances of E are those who are valid in the context of y. In the context of y the bound variable x does not occur, hence it is not allowed in instantiations of E either. If it would be allowed, any x occurring in instances of E would be captured by the $\forall x$ binding structure. This concept is called *variable capture*.

A possible solution to unintended variable capture is to limit the bound variables that may occur in placeholders, by explicitly maintaining for each placeholder a list of bound variables that this placeholder is allowed to refer to. These variables are allowed to be captured, because they exist within the scope of the original occurrence of the placeholder. In the example above, the placeholder E may not refer to x, hence x does not occur in its set of allowed bound variables.

This set of variables can be computed from the initial context in which the placeholders occur in the goal. The set of allowed bound variables for placeholder E, will be referred to as $\mathcal{C}(E)$. For example: our (unsolvable) goal is $(\forall y.(\exists x.P(x,y) \wedge Q(E))) \Rightarrow (\exists z.P(z,z) \wedge Q(z))$. $\mathcal{C}(E) = \{x, y\}$. During the derivation, we apply $\forall - E$ on $(\forall y.(\exists x.P(x,y) \wedge Q(E)))$ using $(y \mapsto E)$. The idea would then be to later replace E by x (which is allowed) to complete the proof. However, the substitution $(y \mapsto E)(\exists x.P(x,y) \wedge Q(E))$, by definition of

substitution, is equal to $(\exists x'.(y \mapsto E)((x \mapsto x')(P(x,y) \wedge Q(E))))$, where x' is a fresh variable. The result is $(\exists x'.P(x', E) \wedge Q(E(x := x')))$. In this case, x' may not be x, since x can occur in E[1]. Hence, replacing E by x is allowed, but results in $(\exists x'.P(x', x) \wedge Q(x'))$, which is not equal to the goal. The variable x in the instance of the placeholder introduced by substitution no longer refers to the original bound variable x. Replacing E by x' would do the trick, but is not allowed, since E may only refer to bound variables x and y.

Definition 1 ($\mathcal{C}$(E,G)). *The set $\mathcal{C}(E, G)$ of bound variables that are allowed to occur in E is dependent on the original goal G in which E occurs. It can be computed as $\mathcal{C}(E, G)$ by:*

$$\begin{array}{ll}
\mathcal{C}(E, \bot) & = \mathcal{V} \\
\mathcal{C}(E, F_1 \Rightarrow F_2) & = \mathcal{C}(E, F_1) \cap \mathcal{C}(E, F_2) \\
\mathcal{C}(E, \forall x.F) & = \{x\} \cup \mathcal{C}(E, F) \\
\mathcal{C}(E, P(t_1, \ldots, t_n)) & = \mathcal{C}(E, t_1) \cap \ldots \cap \mathcal{C}(E, t_n) \\
\mathcal{C}(E, E[v_1, \ldots, v_n := t_1, \ldots, t_n]) & = \emptyset \\
\mathcal{C}(E, G[v_1, \ldots, v_n := t_1, \ldots, t_n]) & = \mathcal{C}(E, t_1) \cap \ldots \cap \mathcal{C}(E, t_n) \\
\mathcal{C}(E, f(t_1, \ldots, t_n)) & = \mathcal{C}(E, t_1) \cap \ldots \cap \mathcal{C}(E, t_n) \\
\mathcal{C}(E, v) & = \mathcal{V}
\end{array}$$

For example $C(E, (\forall x.P(E)) \Rightarrow (\forall y.P(E))) = \emptyset$. Otherwise, replacing E by x would lead to a formula containing $(\forall y.P(x))$, where the x would refer to a bound variable out of scope. Also, $C(E, (\forall x.P(E)) \Rightarrow (\forall y.P(E[x := y]))) = \emptyset$, even though it would seem that $\{x\}$ would lead to correct results. However, this is only true if the x in the explicit substitution is read to mean the bound variable x of the left hand side of the implication. This bound x, however, is not in scope and hence the x in the substitution must be *another* x that is free. This is in accordance with the Barendregt convention: it is not the name of the variable that is important, but the place where it is bound. Free variables are implicitly universally bound and hence, are in scope of all placeholders.

Still, there are formulas with more than one occurrence of E for which $C(E, G)$ is not empty: $C(E, (\forall x.P(E) \Rightarrow Q(E)))$=$\{x\}$.

Lemma 1. *If $x \in C(E, A)$, then E does not occur in A or $x \in BV(A)$.*

Proof. By induction on the structure of A: E does not occur in $\bot$ or v; $x \in BV(\forall x.F)$ and $x \notin \emptyset$.

Definition 2 (Admissibility of replacements). *A replacement σ is called* admissible *for $A_1, \ldots A_n \vdash A$, if for every $E \in Dom(\sigma)$ we have*

$$FV(\sigma(E)) \cap BV(A) \subseteq C(E, A_1 \Rightarrow (\ldots \Rightarrow A_n))$$

[1] x' cannot occur freely in E, since it is fresh. Also, bound variables are different from free variables by definition.

In order to allow substitutions to be mappings from $\mathcal{V}$ to $T_{\mathcal{H}}$, the definition of replacement has to be changed into:

$$\sigma(\forall x.P) = (\forall x.\sigma(P))$$
$$\sigma(H[v_1,\ldots,v_n := t_1,\ldots,t_n]) =\uparrow ([v_1,\ldots,v_n := \sigma(t_1),\ldots,\sigma(t_n)])(\sigma(H))$$

Also, the definition of substitution will be changed a bit. Since the side-condition $y \notin (Dom(\theta) \cup FV(Range(\theta))) \cup (FV(P) \setminus \{x\})$ cannot be decided upon if $Range(\theta)$ contains placeholders, we always have to rename the bound variable to a fresh variable in these cases.

Lemma 2. *Let A and t be terms in the extended signature, let x be a variable and let σ be a replacement. Then $\sigma((x \mapsto t)A) \equiv (x \mapsto \sigma(t))(\sigma(A))$.*

Proof. By induction on the structure of A. In most cases this is trivial. We consider the cases $A \equiv x$, $A \equiv (\forall y.A')$ and $A \equiv H[v_1,\ldots,v_n := t_1,\ldots,t_n]$:

Case $A \equiv x$:
$\sigma((x \mapsto t)x)$
$\equiv$ { *Substitution* }
$\sigma(t)$
$\equiv$ { *Substitution* }
$(x \mapsto \sigma(t))x$
$\equiv$ { *Definition replacement with $x \notin Dom(\sigma)$* }
$(x \mapsto \sigma(t))(\sigma(x))$

Case $A \equiv (\forall y.A')$:
$\sigma((x \mapsto t)(\forall y.A'))$
$\equiv$ { *Substitution* }
$\sigma(\forall y'.(x \mapsto t)((y \mapsto y')A'))$
$\equiv$ { *Replacement* }
$(\forall y'.\sigma((x \mapsto t)((y \mapsto y')A')))$
$\equiv$ { *Induction hypothesis* }
$(\forall y'.(x \mapsto \sigma(t))(\sigma((y \mapsto y')A')))$
$\equiv$ { *Induction hypothesis* }
$(\forall y'.(x \mapsto \sigma(t))((y \mapsto \sigma(y'))(\sigma(A'))))$
$\equiv$ { *Obviously, $y' \notin Dom(\sigma)$* }
$(\forall y'.(x \mapsto \sigma(t))((y \mapsto y')(\sigma(A'))))$
$\equiv$ { *Substitution* }
$(x \mapsto \sigma(t))(\forall y.\sigma(A'))$
$\equiv$ { *Replacement* }
$(x \mapsto \sigma(t))\sigma(\forall y.A')$

Case $A \equiv E(x_1,\ldots,x_n := e_1,\ldots,e_n)$:

$\sigma(x \mapsto t)(E(x_1, \ldots, x_n := e_1, \ldots, e_n))$
$\equiv$ { *Substitution* }
$\sigma(E \downarrow ((x \mapsto t) \circ \uparrow (x_1, \ldots, x_n := e_1, \ldots, e_n)))$
$\equiv$ {• **Case 1:** $x \notin \{x_1, \ldots, x_n\}$ }
$\sigma(E(x_1, \ldots, x_n, x := (x \mapsto t)e_1, \ldots, (x \mapsto t)e_n, t))$
$\equiv$ { *Replacement* }
$\uparrow (x_1 \mapsto \sigma((x \mapsto t)e_1), \ldots, x_n \mapsto \sigma((x \mapsto t)e_n), x \mapsto \sigma(t))\sigma(E)$
$\equiv$ { *Induction hypothesis* }
$\uparrow (x_1 \mapsto ((x \mapsto \sigma(t))\sigma(e_1)), \ldots, x_n \mapsto ((x \mapsto \sigma(t))\sigma(e_n)), x \mapsto \sigma(t))\sigma(E)$
$\equiv$ { *Function composition* }
$(x \mapsto \sigma(t)) \circ \uparrow (x_1 \mapsto \sigma(e_1), \ldots, x_n \mapsto \sigma(e_n))\sigma(E)$
$\equiv$ { *Replacement* }
$(x \mapsto \sigma(t))\sigma(E(x_1, \ldots, x_n := e_1, \ldots, e_n))$

$\equiv$ {• **Case 2:** $x \in \{x_1, \ldots, x_n\}$ }
$\sigma(E(x_1, \ldots, x_n := (x \mapsto t)e_1, \ldots, (x \mapsto t)e_n))$
$\equiv$ { *Replacement* }
$\uparrow (x_1 \mapsto \sigma((x \mapsto t)e_1), \ldots, x_n \mapsto \sigma((x \mapsto t)e_n))\sigma(E)$
$\equiv$ { *Induction hypothesis* }
$\uparrow (x_1 \mapsto ((x \mapsto \sigma(t))\sigma(e_1)), \ldots, x_n \mapsto ((x \mapsto \sigma(t))\sigma(e_n)))\sigma(E)$
$\equiv$ { *Function composition* }
$(x \mapsto \sigma(t)) \circ \uparrow (x_1 \mapsto \sigma(e_1), \ldots, x_n \mapsto \sigma(e_n))\sigma(E)$
$\equiv$ { *Replacement* }
$(x \mapsto \sigma(t))\sigma(E(x_1, \ldots, x_n := e_1, \ldots, e_n)))$

Theorem 2 is a soundness result and shows how derivations with placeholders can be used to obtain derivations in regular first-order logic.

Theorem 2 (Replacements). *if $\Gamma \vdash A$ can be derived and σ is an admissible replacement for $\Gamma \vdash A$, then $\sigma(\Gamma) \vdash \sigma(A)$ can also be derived.*

Proof. By induction on the length of the derivation. Most cases are trivial. We discuss $\forall - E$ and $\forall - I$.

Assume the last step in the derivation was $\forall - E$, then the result can be written as $\Gamma \vdash (x \mapsto t)(A')$, which was derived from $\Gamma \vdash (\forall x.A')$. By induction, we get $\sigma(\Gamma) \vdash \sigma(\forall x.A')$, hence $\sigma(\Gamma) \vdash (\forall x.\sigma(A'))$. Using $\forall - E$, we can derive $\sigma(\Gamma) \vdash (x \mapsto \sigma(t))(\sigma(A'))$, which by lemma 2 is equal to $\sigma(\Gamma) \vdash \sigma((x \mapsto t)A')$.

Assume the last step in the derivation was $\forall - I$, then the result can be written as $\Gamma \vdash (\forall x.A')$, which was derived from $\Gamma \vdash A'$ and the side condition $x \notin FV(\Gamma)$. The induction hypothesis gives us $\sigma(\Gamma) \vdash \sigma(A')$. We need to proof that $x \notin \sigma(FV(\Gamma))$. From the premise we know that $x \notin FV(\Gamma)$. Say $\Gamma = A_1, \ldots A_n$, then we get $x \notin FV(A_i)$ for any i. Assume $x \in FV(\sigma(E))$. Then, since σ is admissible, $x \in C(E, A_1 \Rightarrow (\ldots \Rightarrow A_n))$. By definition of C, we get $x \in$

$C(E, A_1), \ldots, x \in C(E, A_n)$, which by Lemma 1 means that E does not occur in A_i or $x \in BV(A_i)$. If E does not occur in A_i, then $x \notin FV(\sigma(A_i))$. In case $x \in BV(A_i)$ we also get $x \notin FV(\sigma(A_i))$. Hence, $x \notin FV(A_i)$ and therefore $x \notin FV(\Gamma)$. We may conclude that $\sigma(\Gamma) \vdash \sigma(\forall x.A')$ is derivable.

Corollary 1 (Consistency). *$\Gamma \vdash \bot$ is not derivable in the placeholder calculus.*

Proof. Use Theorem 2 and consistency of first-order logic.

Due to theorem 2, it is possible to instantiate (or fill in) placeholders during the construction of a proof. This can be stated as a new instantiate rule in the derivation system, but since we only consider goal directed reasoning, this is hardly helpful. The goal of the proof has been stated in the beginning, using some placeholders to fill in gaps (like expressions of a program that are not yet known). The instantiate rule would then allow us to draw a more specific conclusion based on our derivation, but it would not change the derivation itself.

Instead, instantiating placeholders will be a feature of the implementation. A tool can instantiate all occurrences of a placeholder throughout the proof, hence affecting the entire derivation. Theorem 2 merely claims that this feature is sound. The user of an implementation can now state a goal that is not yet complete and start proving it. During the construction of the proof the placeholder can be filled in according to the needs of the proof. This corresponds exactly to the program derivation example at the beginning of this paper and hence, is exactly the way programs are derived with pen and paper in the Eindhoven style of programming.

Considering the logic this way, we do have an extended syntax, but we do not have a new derivation system. Hence, most probably all meta-theoretical properties of the original derivation system will still hold for our placeholder calculus. After all, every derivation in the extended syntax merely represents all derivations with the same structure abstracted by placeholders. However, one has to be careful, if meta-theorems use premises that claim properties of formulas that do not necessarily hold for placeholders. The constraints placed upon the placeholders only guarantee that the side-conditions of the derivation rules are not broken. They may not be strong enough to guarantee correctness of premises of some meta-theoretical properties.

6 Discussion

Placeholder calculus can be implemented relatively easy into existing theorem provers. The following steps have to be taken:

- The term-structure of the system has to be extended with placeholders and representations for substitutions. The lifting and unlifting functions are

merely needed in the theory. An implementation may use the internal representation of substitutions for both the substitution itself and its syntactic representation. Also, the substitutions are not part of the input language. The system only has to be able to display substitutions for sub-results.

- Once the goal is formulated we have to compute for each placeholder the set of bound variables that it may refer to. This boils down to computing the set of bound variables in the context in which the placeholder occurs. Since the parser for the input language has to compute this set in order to decide if the input is valid, this should be little extra work.

- The system has to be able to perform replacements and check each replacement for validity. Since the definition of replacement with respect to quantified formulas is far less complicated than substitution, this should be easy to implement.

- Once a proof is completed, the system has to check if all placeholders have been filled in. If not, the theorem has been proved for all values that can be given to these placeholders. In such case, one has a proof template rather than a proof. If the system does not want to accept these proofs, it can fill in an arbitrary value for the remaining placeholders.

In literature, placeholders are often referred to as Meta-Variables [6, 10]. However, Meta-variables is not a correct name: the variables are first-order citizens and hence, are part of the logic itself; not its meta-theory. Therefore we choose to call them placeholders, which is exactly what they are: placeholders for parts of the formula that have yet to be filled in.

This calculus does not need an explicit substitution rule (or β-conversion rule), since the substitutions are applied automatically by definition whenever possible. As soon as a placeholder is filled in, any pending substitutions on its value are performed due to the definitions of replacement, lifting and unlifting.

The logic presented in this paper is also applicable to the family of Boyer-Moore theorem provers, since a lambda term representing the proof is not required.

Usually several goals exist in several contexts (i.e. one attempts to solve a set of (sub)judgements) and these are mutually connected by the same placeholders. In such case, filling in a placeholder must be done in all (sub)goals simultaneously.

In type theory, placeholders are typed; hence it is known what may be filled in for each placeholder. Since the placeholders cannot be typed in the systems we consider, we used the non-terminals of the logic's grammar to distinguish between several kinds of placeholders. This is somewhat weaker than specifying the type. Hence, it creates more freedom on the one hand, but restricts its application on the other hand.

References

1. Edsger W. Dijkstra, *Guarded commands, nondeterminacy and formal derivation of programs*, Commun. ACM **18** (1975), no. 8, 453–457.
2. Edsger W. Dijkstra, *A discipline of programming*, Prentice-Hall International, 1976.
3. M. Franssen, *Cocktail: A tool for deriving correct programs*, Ph.D. thesis, Eindhoven University of Technology, 2000.
4. Michael Franssen, *Embedding first-order tableaux into a pure type system*, Electronic Notes in Theoretical Computer Science (Didier Galmiche, ed.), vol. 17, Elsevier Science Publishers, 2000.
5. Murdoch J. Gabbay and Aad Mathijssen, *One-and-a-halfth-order logic*, Journal of Logic and Computation (2007).
6. Herman Geuvers and Gueorgui I. Jojgov, *Open proofs and open terms: A basis for interactive logic*, CSL '02: Proceedings of the 16th International Workshop and 11th Annual Conference of the EACSL on Computer Science Logic (London, UK), Springer-Verlag, 2002, pp. 537–552.
7. C. A. R. Hoare, *An axiomatic basis for computer programming*, Commun. ACM **12** (1969), no. 10, 576–580.
8. Anne Kaldewaij, *Programming: the derivation of algorithms*, Prentice-Hall international series in Computer Science, Prentice Hall, 1990.
9. T. Laan and M. Franssen, *Embedding first-order logic in a pure type system with parameters*, Journal of Logic and Computation **11** (2001), no. 4, 545–557.
10. C. Muñoz, *Dependent types with explicit substitutions: A metatheoretical development*, International Workshop TYPES'96, LNCS, vol. 1512, Springer-Verlag, 1996, pp. 294–316.