Completeness of Resolution for Definite Answers

Tanel Tammet

e-mail: tammet@cs.chalmers.se

A slightly modified version of this report will appear in the
Journal of Logic and Computation.

Printed as: Programming Methodology Group report 79, April 1994

Department of Computing Science
Chalmers University of Technology and
Göteborg University
S-41296 Göteborg, Sweden

Completeness of Resolution for Definite Answers

Tanel Tammet
Department of Computing Science
Chalmers University of Technology and Göteborg University
S-41296 Göteborg, Sweden
e-mail: tammet@cs.chalmers.se

Abstract

We investigate the problem of finding a computable witness for the existential quantifier in a formula of the classical first-order predicate logic. The A-resolution calculus based on the program derivation algorithm A of C-L. Chang, R. C-T. Lee and R.Waldinger is used for finding a definite substitution t for an existentially bound variable y in some formula F, such that $F\{t/y\}$ is provable. The term t is built of the function and predicate symbols in F, plus Boolean functions and a case splitting function *if*, defined in the standard way: $if(True, x, y) = x$ and $if(False, x, y) = y$.

We prove that the A-resolution calculus is complete in the following sense: if such a definite substitution exists, then the A-calculus derives a clause giving such a substitution. The result is strengthened by allowing the usage of liftable criterias R of a certain type, prohibiting the derivation of the substitution terms t for which $R(t)$ fails. This enables us to specify, for example, that the substitution t must be in some special signature or must be type-correct, without losing completeness.

Keywords: program derivation, automated theorem proving, resolution, completeness, formal specifications.

1 Introduction

The motivation for this work is to devise efficient automated theorem proving strategies for the first-order theorem proving tasks arising in the formal derivation of programs from specifications. The specific aim of the paper is to present completeness results for certain simple relatively well-known program synthesis algorithms.

One of the standard approaches to automated program construction is using intuitionistic logic with a suitable realizability interpretation to derive programs from proofs (see [5], [11], [8]). The programs derived in this way always enjoy an intuitionistic correctness proof.

Another approach (see [2], [6], [1]) is to use classical logic instead, with the additional restrictions guaranteeing that the proof contains a single definite substitution t into a certain existentially bound variable, and this t is furthermore in a signature where all the function and predicate symbols are assumed to represent computable functions. The derived programs thus always have a classical correctness proof, although they may lack an intuitionistic one.

The following summarizes our motivation for using the second approach (classical logic) for program construction:

- The known realizability interpretations for intuitionistic logic often give programs which contain computationally irrelevant parts. For example, the realization of the following formula

  $$\forall x \exists y (x = y \,\&\, y = x)$$

  is a term $\lambda x.p(x, p(id, id))$ where id is a realization of identity and p is a pair constructor. The A-resolution gives a term $\lambda x.x$ as a program to compute y.

- Some formulas which admit a proof by A-resolution (and hence give a program) are not provable by intuitionistic logic. For example, A-resolution gives a program $\lambda x.x$ for computing y for both of the following formulas, while none of these formulas is intuitionistically provable:

  $$\forall x \exists y ((A \vee \neg A) \,\&\, y = x)$$

  $$(\forall x((0 < x) \Rightarrow (p(x) < x))) \Rightarrow (\forall x((0 \not< x) \vee \exists y(y < x)))$$

- The resolution method with Skolemization and/or conversion to a conjunctive normal form (CNF) cannot be used for intuitionistic logic, although there exist special resolution methods without Skolemization and CNF ([9], [10]) and a tableaux method with partial dynamic Skolemization ([15]) for intuitionistic logic.

- There is a sizeable amount of theory for the resolution method, including different search strategies, decidable classes, etc, which can be used for program derivation by A-resolution.

- The technology developed for automated theorem proving in classical logic can be directly put to use for program derivation by using A-resolution.

1.1 Basic definitions

We consider closed formulas in the first-order predicate logic language with function symbols. When we say that "a formula F is derivable (provable)", we mean that F is derivable in the classical first-order logic, unless we explicitly state otherwise.

We will restrict us to formulas which contain at least one positive occurrence of the existential quantifier or at least one negative occurrence of the universal quantifier. The polarity of subformula occurrence is defined in the standard way:

Definition *A subformula G of a formula F containing only the propositional connectives in $\{\neg, \Rightarrow, \vee, \&\}$ is said to occur positively in F iff it is under an even number of connectives $\neg$ and the left arguments of $\Rightarrow$. G is said to occur negatively iff it is under an odd number of connectives $\neg$ and the left arguments of $\Rightarrow$.*

Instead of speaking about the positive and negative occurrences of quantifiers we will refer to both the positive occurrences of existential quantifiers and negative occurrences of universal quantifiers as *essentially existential quantifiers* and the positive occurrences of universal quantifiers and negative occurrences of existential quantifiers as *essentially universal quantifiers*. We will further simplify the presentation by assuming that each occurence of a quantifier in a formula binds a variable different from all the variables bound by the other quantifier occurrences. If a variable is bound by the essentially existential quantifier, we will say that it is an *existential variable*, otherwise it will be said to be a *universal variable* (since our formulas are assumed to be closed, we prefer not to use the notion of a "free variable" here). We will treat constant symbols (function symbols with the arity 0) as being different from the universal variables, although this difference is important only regarding Skolemization.

As our goal is to derive programs by finding a certain definite substitution t into one of the existential variables of the formula, we assume that our formulas have an associated marker for this specific existential variable, which will be called the *main variable* of the formula. We require that the quantifier occurence Q binding the main variable must be out of the scope of other essentially existential quantifiers. The set of universal variables bound by the set S of all occurrences of the essentially universal quantifiers such that Q is in the scope of all the elements of S is called the set of *parametric variables* of the formula.

Given a formula F and its main variable y, we are looking for a proof of F such that this proof would give a term t for computing a value r for y for any set of values $t_1, t_2, \ldots, t_n$ assigned to parametric variables $x_1, x_2, \ldots, x_n$ so that a substitution instance $F\{t_1/x_1, t_2/x_2, \ldots t_n/x_n, r/y\}$ of the formula F would be provable in the first-order classical logic. Here and elsewhere $\{t_1/x_1, \ldots, t_n/x_n\}$ represents the substitution of each t_i $(1 \leq i \leq n)$ for the variable x_i, respectively.

The computable terms t we are looking for are assumed to contain only the function and predicate symbols and parametric variables of F, plus Boolean functions and a case-analysis function "*if*" defined in the standard way: $if(True, x, y) = x$ and $if(False, x, y) = y$. Since not all the predicate and function symbols in F necessarily represent computable functions, the signature of t may be further restricted to a subset of function and predicate symbols of F, representing computable functions.

The proof search is carried out in a modified resolution calculus.

2 The resolution calculus

Since its introduction in [13] the resolution method has been the most popular method of automated theorem proving in classical predicate logic. A large body of

theory concerning various modifications and special strategies of resolution has been developed. First we will define the standard notions of the resolution method. For further details see, for example, [1] or [3].

The Skolemization algorithm transforms any formula F to a formula F' such that all the variables in F' are assumed to be bound by the essentially existential quantifiers, F' contains no quantifiers and F is provable iff F' is provable.

Definition *The Skolemization algorithm: The formula F is transformed to the formula F_1 in the following way: each universal variable x in F bound by a quantifier G is replaced by a term t (called a Skolem term) consisting of a new function symbol f (called a Skolem function) applied to argument variables $y_1, \ldots, y_n$ (where $0 \leq n$) being the sequence of all variables bound by the essentially existential quantifiers $(Qy_1), \ldots, (Qy_n)$ in F such that G is in the scope of exactly these essentially existential quantifiers and each (Qy_i) is exactly in the scope of essentially existential quantifiers $(Qy_1), \ldots, (Qy_{i-1})$. The result of Skolemization is the formula F' obtained from F_1 by removing all the quantifiers in the formula F_1.*

Definition *A* term *is either a variable, a constant symbol or a nonconstant function symbol applied to one or more terms. In certain cases we extend the notion of the term to contain also literals, Boolean functions and the case analysis function if. In these cases we either say explicitly that the term may contain literals or this is clear from the context.*

Definition *An* atom *is a predicate symbol applied to zero or more terms. A* literal *is either an atom or an atom preceded by the negation sign.*

Definition *A* clause *is a finite set of literals. Any clause $\{L_1, L_2, \ldots, L_n\}$ is interpreted as the disjunction $L_1 \vee L_2 \vee \ldots \vee L_n$ of its member literals. All variables in a clause are interpreted as being quantified by the universal quantifier $\forall$.*

Definition *The* clause form *of a formula F is the set of clauses $\{C_1, C_2, \ldots, C_m\}$ obtained from the Skolem form F' of F by converting $\neg F'$ to a conjunctive normal form by equivalent transformations. Any such clause set $\{C_1, C_2, \ldots, C_m\}$ is interpreted as a conjunction $C_1 \,\&\, C_2 \,\&\, \ldots \,\&\, C_m$ of its member clauses.*

Definition *Consider a formula F with parametric variables $x_1, \ldots, x_n$ and the Skolemized form F' (or a clause form S) of F. The Skolem constants $c_1, \ldots, c_n$ in F' or S replacing the parametric variables in F are called* parametric Skolem constants *of F' or S.*

Any formula of classical logic can be converted to the clausal form. A formula of classical logic is provable iff its clausal form in unsatisfiable.

New clauses are derived by the rules of binary resolution and factorization. Before each resolution step, all the variables in one of the resolved clauses have to be

renamed, so that the two clauses will be variable disjoint. We denote by $mgu(L, L')$ the most-general unifier of the terms or literals L and L'.

The resolution rule is defined as:

$$\frac{\{L, \Gamma\} \quad \{\neg L', \Delta\}}{\{\Gamma, \Delta\}\sigma}\sigma = mgu(L, L')$$

provided that the atoms L and L' are unifiable. The literals L and $\neg L'$ are called the *literals resolved upon.*

The factorization rule is defined as:

$$\frac{\{L, L', \Gamma\}}{\{L, \Gamma\}\sigma}\sigma = mgu(L, L')$$

provided that the literals L and L' are unifiable.

Definition *For a clause set S we define $Fact(S)$ as the set of factors of S and $Res(S)$ as the set of resolvents of S and all factors of these resolvents. Additionally we then define:*

$$\mathcal{R}^0(S) = Fact(S),$$
$$\mathcal{R}^{i+1}(S) = \mathcal{R}^i(S) \cup Res(\mathcal{R}^i(S)) and$$
$$\mathcal{R}^*(S) = \bigcup_i \mathcal{R}^i(S)$$

We say that a clause C is derivable from *a clause set S iff $C \in \mathcal{R}^*(S)$.*

The completeness theorem for resolution states that a set of clauses S is unsatisfiable (i.e. has no model) if and only if an empty clause (i.e. contradiction) is derivable from S.

3 ANS-method and the D-calculus

3.1 ANS-method

Given a Skolemized formula F, there is a well-known method (we will call it ANS-method) for finding a finite set of substitutions applied to some variable y during the refutation. Our presentation of the ANS-method differs slightly from the presentation in [4] and [1].

Definition *A formula, clause set, clause, literal or a term is called* ground *iff it does not contain any variables.*

Definition *An* answer clause *is either an empty clause or a clause containing only literals with the special predicate A called the* answer predicate.

Definition ANS-method: *Given a clause set S (S is assumed not to contain the predicate A) and a variable y, a new clause set S' is formed by adding a new literal $A(y)$ with an answer predicate A to each clause in S containing the variable y. The refutation of S is found iff an answer clause C is derived from S. In case C is empty, no substitution has been applied to y and $S\{t/y\}$ is unsatisfiable for arbitrary ground t. In case $C = \{A(t_1), \ldots, A(t_n)\}$ (where $1 \leq n$), the set of substitutions for y is $\{t_1, \ldots, t_n\}$, thus the clause set $S\{t'_1/y\} \cup \ldots \cup S\{t'_n/y\}$ is unsatisfiable, where $t'_1, \ldots, t'_n$ are arbitrary ground instances of $t_1, \ldots, t_n$, respectively.*

For the correctness and completeness proofs of the ANS-method see [4] or [1].

Example Consider the formula F:

$$(P(a) \vee P(c)) \Rightarrow \exists y P(y)$$

and the main variable y in F. Skolemization gives $(P(a) \vee P(c)) \Rightarrow P(y)$. The clause form S of F: $\{\{P(a),\ P(c)\}, \{\neg P(y)\}\}$. The result of adding the answer literals A is the clause set S':

$$\{\{P(a),\ P(c)\}, \{\neg P(y),\ A(y)\}\}$$

Resolution derives the answer clause $\{A(a), A(c)\}$ from S', thus the set of substitutions for y is $\{a, c\}$.

There is a well-known class of formulas where the ANS-method always gives a single substitution for any unsatisfiable set of clauses: namely, the Horn Class, which is the foundation of the Prolog programming language. A Prolog inference engine for queries containing variables may be seen as a special case of the ANS-method.

In the general case, however, the set of substitutions computed by the ANS-method may contain several elements and there is no direct way to use this set as a program for finding a single definite substitution t into the main variable of a formula.

Example Consider the formula F from the previous example and the variable y in F. In the previous example the ANS-method computed the set $S = \{a, c\}$ for the Skolemized version of F. There is no way to select a definite term r from S so that $((P(a) \vee P(c)) \Rightarrow P(y))\{r/y\}$ would be provable.

Example Consider the formula $R \doteq (((P(a) \vee P(c)) \,\&\, P(d)) \Rightarrow \exists y P(y))$ and the main variable y in R. The ANS-method may return a set $\{a, c\}$ for y. However, there exists a possible result set $\{d\}$ of the ANS-method, containing a single element d, such that the formula $(((P(a) \vee P(c)) \,\&\, P(d)) \Rightarrow P(y))\{d/y\}$ is provable.

3.2 D-calculus

The D-calculus is used for finding a definite substitution t for the main variable of a formula F, such that t is built of the function symbols in F. The D-calculus is a weaker version of the forthcoming A-calculus, with the difference being in that the A-calculus allows the substitution term t to contain the case splitting function "*if*", Boolean functions and the predicate symbols in F, whereas the D-calculus does not.

Definition *The D-calculus is obtained from the ordinary resolution calculus by prohibiting the resolution rule to be applied to two premisses both containing the answer predicate A and adding the new D-resolution rule for this case:*

The D-resolution rule:

$$\frac{\{L, A(t), \Gamma\} \quad \{\neg L', A(g), \Delta\}}{\{A(t), \Gamma, \Delta\}\sigma\rho}\sigma = mgu(L, L'), \rho = mgu(t\sigma, g\sigma)$$

where $A(t)$ and $A(g)$ are answer literals, on the condition that both the atoms L and L' are unifiable, as well as the terms $t\sigma$ and $g\sigma$.

Definition *A* definite answer clause *is either an empty clause or an answer clause containing a single literal.*

Proof search by the D-calculus is completed iff a definite answer clause is found. Notice that in case no clause in a clause set S contains more than one answer literal, in D-calculus it is impossible to derive from S a clause containing more than one answer literal.

Definition *By* D-completeness *of a certain resolution calculus C we will mean completeness for definite answers: if there exists a ground term t such that a substitution instance $S\{t/y\}$ of a clause set S is unsatisfiable, the calculus C will either derive an empty clause or a clause $\{A(g)\}$ from the clause set S' obtained from S by adding a literal $A(y)$ to every clause in S containing y, such that $S\{g'/y\}$ is unsatisfiable for any ground instance g' of g.*

The analogue of the following D-completeness lemma is proved in [14]. We give our own proof with the principal idea similar to the idea of the proof of the forthcoming theorem 2 (the main result of the paper).

Lemma 1 (D-completeness of the D-calculus) *Let G be a clause set containing a variable y. Suppose that the clause set $G\{t/y\}$ is unsatisfiable for some ground term t. Let G' be a clause set obtained from G by adding an answer literal $A(y)$ to each clause of G containing the variable y. Then the D-calculus will derive a definite answer clause from G'.*

Proof Since we assume that $G\{t/y\}$ is unsatisfiable for the term t, the clause set $G'' \doteq G' \cup \{\neg A(t)\}$ is also unsatisfiable, since all the n clauses in $G\{t/y\}$ and not

in G'' can be derived from G'' by n binary resolution steps with the clause $\{\neg A(t)\}$ and all the clauses in G' containing the literal $A(y)$.

We will show that since G'' is unsatisfiable, the ordinary resolution calculus will derive a definite answer clause from G''.

Consider the resolution method restricted by the following ordering strategy (a special case of semantic resolution or lock resolution). For any clause C containing a literal L with the predicate symbol A, L may be resolved upon in C only if all literals in C have a predicate symbol A. Such a restricted resolution method is complete (see [1] or [3]) for any predicate symbol A.

Any deduction of an empty clause from G'' either does not involve any resolution steps with the clauses containing the predicate symbol A or the resolution steps upon literals with the symbol A are the last m steps of the deduction with the clause $\{\neg A(t)\}$ as one of the premisses for each step:

$C_1 = \{A(l_1), \ldots, A(l_m)\}$

$C_2 = \{A(l_2), \ldots, A(l_m)\} mgu(t, l_1)$

$C_3 = \{A(l_3), \ldots, A(l_m)\} mgu(t, l_1) mgu(t, l_2)$

$\ldots$

$C_m = \{A(l_m)\} mgu(t, l_1) mgu(t, l_2) \ldots mgu(t, l_{m-1})$

$C_{m+1} = \{\}$

In the first case the empty clause is derivable also from G'.

In the second case the set of all literals in the clause C_1 is unifiable with a most-general unifier τ so that $A(l_1)\tau = A(l_2)\tau = \ldots = A(l_m)\tau$ and $t = l_1\tau\mu$ for some substitution μ. Therefore factorization derives a clause $\{A(l_1)\}\tau$ from C_1. As C_1 was derived without any steps involving the clause $\{\neg A(t)\}$, the definite answer clause $\{A(l_1)\}\tau$ is derivable from G'.

Thus, in both cases the definite answer clause is derivable from the clause set G'. $\square$

Example Consider a formula R:

$$((P(a) \vee P(c)) \,\&\, P(d)) \Rightarrow P(y)$$

with the main variable y. The clause form G' of R after adding the answer literals is:

$$\{\{P(a), P(c)\}, \{P(d)\}, \{\neg P(y), A(y)\}\}.$$

D-calculus cannot derive an answer clause $\{A(a), A(c)\}$ from G', but it does derive a definite answer clause $\{A(d)\}$.

Consider a formula F with the main variable y and the parametric variables $x_1, \ldots, x_n$. Let G be the clause form of F. Suppose there exists a ground term t such that $G\{t/y\}$ is unsatisfiable (as previously shown, if such a t exists, it can be found by the D-calculus).

Our main aim is to find a program for computing a value r for y for any set of values $t_1, \ldots, t_n$ assigned to $x_1, \ldots, x_n$ so that a substitution instance $F\{t_1/x_1, \ldots, t_n/x_n, r/y\}$ would be provable. The possible presence of Skolem functions in t poses problems for using the term t directly as r.

Let the Skolem constants $c_1, \ldots, c_n$ replace the parametric variables $x_1, \ldots, x_n$ in G. We will call such Skolem constants *parametric Skolem constants*. We will build a term $g \doteq \lambda x_1, \ldots, x_n.t'$ where t' is obtained by replacing the constant symbols $c_1, \ldots, c_n$ in t by the variables $x_1, \ldots, x_n$, respectively, and use the term g as a program for computing the required value r. Indeed, for the terms given by the D-calculus the program g does nothing but substitute the values of argument variables into its body.

The clause set G may also contain other Skolem functions besides the parametric Skolem constants $c_1, \ldots, c_n$. Let S be the set of all Skolem functions in G. If t contains any Skolem functions in $S - \{c_1, \ldots, c_n\}$, then t cannot generally be used for computing a required value r for y.

Example Consider the provable formula F with the main variable y and the single parametric variable x:

$$(\exists z P(z)) \Rightarrow \forall x \exists y (P(g(x))) \Rightarrow P(y))$$

The Skolemized form F' of F is $(P(c_2) \Rightarrow (P(g(c_1)) \Rightarrow P(y)))$, where c_1 is a parametric and c_2 is a nonparametric Skolem constant. The clause form G' of F after adding the answer-literals is $\{\{P(c_2)\}, \{P(g(c_1))\}, \{\neg P(y), A(y)\}\}$. The D-calculus derives a definite answer clause $\{A(c_2)\}$ from G', but $\{c_2\}$ cannot be used as a program for computing a value of y, since it contains the nonparametric Skolem constant c_2. Indeed, the formula

$$(\exists z P(z)) \Rightarrow (P(g(s)) \Rightarrow P(c_2))$$

is not provable for any s.

However, the D-calculus also derives a definite answer-clause $\{A(g(c_1))\}$ which does not contain nonparametric Skolem functions. Thus $g(c_1)$ is usable for computing a value of y, i.e. the formula $(\exists z P(z)) \Rightarrow (P(g(s)) \Rightarrow P(g(s)))$ is provable for any s.

Definition *A formula F belongs to a* Simple Class *iff it contains no essentially universal quantifiers except the ones binding the parametric variables.*

As the Skolemized forms of formulas in the Simple Class do not contain any nonparametric Skolem functions, any term given by the D-calculus for the main variable of these formulas can be used as a program.

In the general case we can ensure usability of the terms in the derived definite answer clauses by using the following restricted form of the D-calculus.

Definition *A computable predicate R on (possibly non-ground) terms is called a* liftable term restriction *iff it has the following property:*

$$\forall t \forall \sigma . R(t\sigma) \Rightarrow R(t)$$

where t is a term and σ is a substitution.

Example Consider a set Σ of function symbols. Let Σ' be the set of typed function symbols formed from Σ by assigning each function symbol in Σ a certain monomorphic (without type variables) type. Let $R_T(t)$ be defined on the terms in the signature Σ as "t is type-correct in the typed signature Σ'". Let R_T be always false on terms outside the signature Σ. It is well-known that such an R_T is indeed decidable. It can be easily seen that R_T is a liftable term restriction.

Definition D(R)-calculus *is obtained from the D-calculus by the following restriction: it is prohibited to derive any clause C containing an answer literal $A(t)$ such that $R(t)$ does not hold for a given liftable term restriction R.*

For example, we can define a certain liftable term restriction $R_\Sigma(t)$ as "t does not contain function symbols from the set Σ". It is easily seen that for any Σ R_Σ is indeed a liftable term restriction.

The set of nonparametric Skolem functions in a clause set can be taken as the set Σ. In the last example we take $\Sigma = \{f\}$.

The definition of D(R)-completeness is obtained from the definition of D-completeness by requiring that the terms t, g and g' satisfy the criteria R.

D(R)-completeness of the D(R)-calculus is proven in the same way as D-completeness is proven for the D-calculus.

Lemma 2 *Let G be a clause set containing a variable y. Let R be a liftable term restriction. Suppose that the clause set $G\{t/y\}$ is unsatisfiable for some ground term t such that $R(t)$ holds. Let G' be a clause set obtained from G by adding an answer literal $A(y)$ to each clause of G containing the variable y. Then the D(R)-calculus will derive a definite answer clause C from G' such that C is either empty or has a form $A(g)$ such that $R(g)$ holds.*

Proof The proof is obtained from the D-completeness proof of the D-calculus by requiring that $R(t)$ holds and noticing that if $t = l_1\tau\mu$ for some term l_1 and some substitutions τ and μ, then due to the liftability of R also $R(l_1\tau)$ holds. □

Definition *Given a clause set S and a liftable term restriction R, we say that S is* refutable by the D(R)-calculus *iff D(R)-calculus derives a definite answer clause C from S such that either C is empty or has a form $A(g)$ such that $R(g)$ holds.*

Definition *Given a formula F with the main variable y, we say that F* is provable by the D(R)-calculus *iff G' is refutable by the D(R)-calculus, where G' is obtained from the Skolem form G of F by adding answer literals $A(y)$ to all the clauses in G containing the variable y and $R(t)$ is defined as "t does not contain nonparametric Skolem functions in G".*

4 A(R)-calculus

The D(R)-calculus fails to find a proof for a large class of formulas which admit a proof in intuitionistic logic. The reason for this, roughly speaking, is that intuitionistic logic assumes subformulas of any formula F to have an associated program (realization of the formula), whereas the D(R)-calculus assumes only the function symbols in F to have an associated program.

Example Consider a formula F:

$$(P(a) \vee P(b)) \Rightarrow \exists y P(y)$$

F does not admit a proof by the D(R)-calculus. However, F is provable in the intuitionistic logic.

We argue that in program specifications the following restriction (stemming from the A(R)-calculus) is natural: nonatomic formulas are never assumed to have associated programs (realizations), only some predicate and function symbols and variables may be assumed to have associated programs. Thus nonatomic formulas composed of atomic formulas by propositional connectives may only have a derived associated program. For example, if a predicate P has an associated program p, then a formula $P(x) \vee P(y)$ has a derived associated program $\lambda xy.Or(p(x), p(y))$ where Or is a standard Boolean disjunction.

The A(R)-calculus we give (it is based on the A-resolution calculus presented in [1]) is an extension of the D(R)-calculus which allows the derivation of answer literals containing the case analysis function *if* and predicate symbols. The function *if* is defined in the standard way: $if(True, x, y) = x$ and $if(False, x, y) = y$. The first arguments of the *if*-terms are assumed to be type-correct (according to some given computable type-checking algorithm) literals containing only computable function and predicate symbols.

Showing type-correctness of literals and computability of function and predicate symbols is outside the scope of the A(R)-calculus. For the purposes of the A(R)-calculus the "type-correctness and computability of a term t (possibly containing *if* and predicate symbols)" means just that $R(t)$ holds for an explicitly given term restriction R. Observe that the check for a first-order term to be type-correct for some monomorphic type assignment to function and predicate symbols is indeed a liftable term restriction.

The notion of a liftable term restriction has to be strengthened in order for the forthcoming special completeness theorem 2 to succeed. The problem is that when a term t satisfying R has a form $if(L, g, h)$ for a literal L and terms g and h, we need both g and h to satisfy R also.

Definition *A liftable term restriction R is called* strongly liftable *iff*

$$\forall t(R(t) \Rightarrow \forall g(iarg(g, t) \Rightarrow R(g)))$$

where $iarg(g,t)$ is true iff g is an argument of some occurrence of the function if in t.

Definition *Given a strongly liftable term restriction R, the* A(R)-calculus *is obtained from the D(R)-calculus by adding a new A-resolution rule:*

$$\frac{\{L, A(t), \Gamma\} \quad \{\neg L', A(g), \Delta\}}{\{A(if(L,g,t)), \Gamma, \Delta\}\sigma}\sigma = mgu(L, L')$$

where $A(t)$ and $A(g)$ are answer literals, on the condition that the atoms L and L' are unifiable and $R(if(L,g,t)\sigma)$ holds.

The A(R)-calculus can be used for finding programs for the main variables of formulas in the same way as the D(R)-calculus.

Example Consider a formula F:

$$(P(a) \vee P(b)) \Rightarrow \exists y P(y)$$

with the main variable y, and let the term restriction R hold for all terms (thus we assume P, a and b be computable). The clause form G' of F (after adding the answer literals):

1) $\{P(a),\ P(b)\}$

2) $\{\neg P(y),\ A(y)\}$

The derivation of the answer clause in the A(R)-calculus:

1 and 2 give 3) $\{P(b),\ A(a)\}$

3 and 2 give an answer clause: $\{A(if(P(b), b, a))\}$

Thus the A(R)-calculus gives a program (without arguments) $if(P(b), b, a)$ for computing the value of y.

Suppose that the algorithm we have for computing the predicate P is defined only on a. In that case we define the restriction $R(t)$ as "any subterm of t with the leading symbol P has either a form $P(a)$ or $P(x)$ for some variable x". Then the A(R)-calculus cannot derive the answer clause $\{A(if(P(b), b, a))\}$; the only answer clause it can derive is $\{A(if(P(a), a, b))\}$.

4.1 $D_{if}(R)$-completeness of the A(R)-calculus

Definition *A type* Boolean *is the set of two logical constants True and False. A* Boolean function *is a function taking $n (0 \leq n)$ arguments of the* Boolean *type and returning a value of the* Boolean *type. In order to simplify the presentation we consider "if" to be a polymorphic function in the sense that exactly these occurrences of "if" are considered to be Boolean for which all the arguments are Boolean.*

Definition *A term t is of a* Boolean type *if either t is a logical constant, literal, or has a form $f(t_1, \ldots, t_n)$ where f is a boolean function and all the t_i $(1 \leq i \leq n)$ are of the Boolean type.*

Notice that we do not allow variables to be of the Boolean type.

Definition *A term t is said to be* B-correct *iff both of the following hold:*

- *each proper subterm s of t which is of a Boolean type is either a first argument of some if-term or is an argument of some Boolean function in t.*
- *the first argument of each if-term in t and all arguments of all Boolean functions in t are of a Boolean type.*

We will start by defining the notion of $D_{if}(R)$-completeness. The definition is obtained from the definition of $D(R)$-completeness by introducing a few changes.

We have to take account of the fact that the term t in the definite answer clause $\{A(t)\}$ derived by the A(R)-calculus may contain predicate symbols and the special function *if* . Given a term t containing *if* and predicate symbols along with a clause set S and the main variable y we cannot any more speak about the satisfiability of the construction $S\{t/y\}$, since $S\{t/y\}$ is not a clause set in the standard sense.

Consider the clause set $S \doteq \{P(a), P(b)\}\,\{\neg P(y)\}$ from the last example. Let $\sigma \doteq \{if(P(a), a, b)/y\}$. Then $S\sigma$ would be $S' \doteq \{P(a), P(b)\}\,\{\neg P(if(P(a), a, b))\}$, for which the standard notion of satisfiability of clause sets is not directly usable.

We will overcome the problem by extending the standard model-theoretic definition of satisfiability for classical first-order predicate logic by defining the semantic function for *if* : $if(True, x, y) = x$ and $if(False, x, y) = y$. Here and in the following we will assume that all the terms in a clause set or a formula we have are B-correct. Thus the semantic value of a first argument of *if* can be only *True* or *False*. We will extend the notion of a clause set by allowing literals to contain B-correct terms containing the function *if* , predicate symbols and Boolean functions.

Definition *By* $D_{if}(R)$-completeness *of a certain resolution calculus C we will mean the following: if there exists a B-correct ground term t such that $R(t)$ holds and the clause set $S\{t/y\}$ is unsatisfiable, then the calculus C will either derive an empty clause or a clause $\{A(g)\}$ from the clause set S' obtained from S by adding an answer literal $A(y)$ to every clause in S containing y, such that g is B-correct, $R(g)$ holds and $S\{g\sigma/y\}$ is unsatisfiable for any ground instance $g\sigma$ of g such that $R(g\sigma)$ holds. R is assumed to be a strongly liftable term restriction and σ is assumed not to contain the function if , Boolean functions or any predicate symbols.*

We will present an explicit algorithm for converting constructions containing *if*-terms to equivalent standard first-order formulas. The algorithm is needed for the forthcoming completeness proof.

Definition *The algorithm A_I takes a term t possibly containing Boolean functions and replaces all the subterms built by Boolean functions (except if and a negation with an atomic argument) by the equivalent terms containing if, True and False instead of Boolean functions. A_I can be easily built by using truth-value matrices of Boolean functions. We skip the details here.*

We could alternatively require A_I to remove also the negation function, but since there is no need for this in the following, we prefer to retain a negation function applied to atoms.

Definition *The algorithm A_T. First, a function step is defined as:*

$$step(g(x_1, \ldots, x_i, if(y_1, y_2, y_3), x_{i+2}, \ldots, x_n) \rightarrow$$

$$if(y_1, g(x_1, \ldots, x_i, y_2, x_{i+2}, \ldots, x_n), g(x_1, \ldots, x_i, y_3, x_{i+2}, \ldots, x_n))$$

$$step(if(if(y_1, y_2, y_3), x_2, x_3)) \rightarrow if(y_1, if(y_2, x_2, x_3), if(y_3, x_2, x_3))$$

for all i $(1 \leq i \leq n)$, for all n and for all predicate and function symbols g except if.

The algorithm A_T takes an arbitrary B-correct term t and computes a treeified form *t' of t by repeated applications of the function step to the term t'' computed by $A_I(t)$*

We say that a term t is in the treeified form *if t does not contain boolean functions except negation applied to atoms and step cannot be applied to t.*

Any B-correct term t has an equivalent treeified form t'.

Definition *The algorithm A_F. First, a function fstep is defined as:*

$$fstep(if(True, y_2, y_3)) \rightarrow y_2$$

$$fstep(if(False, y_2, y_3)) \rightarrow y_3$$

$$fstep(if(y_1, y_2, y_3)) \rightarrow ((y_1 \Rightarrow y_2) \,\&\, (\neg y1 \Rightarrow y3))$$

The algorithm A_F takes an arbitrary B-correct term t of a Boolean type and computes a flattened form *t' of t by repeated applications of the function fstep to the treeified term t'' computed by $A_T(t)$.*

Again, any B-correct term t of a Boolean type has an equivalent treeified form t'.

Example Let t be the term

$$r(x, y) = f(if((P(x) \vee P(y)), g(x, y), h(x, y)))$$

$A_I(t)$ returns t', the term

$$r(x, y) = f(if(if(P(x), True, if(P(y), True, False)), g(x, y), h(x, y)))$$

$A_T(t')$ returns t'', the term

$$
\begin{array}{l}
if(\ P(x), \\
\quad if(True, r(x,y) = f(g(x,y)), r(x,y) = f(h(x,y))), \\
\quad if(\ P(y), \\
\qquad if(True, r(x,y) = f(g(x,y)), r(x,y) = f(h(x,y))), \\
\qquad if(False, r(x,y) = f(g(x,y)), r(x,y) = f(h(x,y)))))
\end{array}
$$

$A_F(t'')$ returns the formula

$$(P(x) \Rightarrow r(x,y) = f(g(x,y))) \ \&$$

$$(\neg P(x) \Rightarrow ((P(y) \Rightarrow r(x,y) = f(g(x,y))) \,\&\, (\neg P(y) \Rightarrow r(x,y) = f(h(x,y)))))$$

We will introduce constructions $S[t/y]$ and $F[t/y]$ for clause sets and formulas, similar to ordinary substitutions $S\{t/y\}$ and $F\{t/y\}$. The difference is that in the newly introduced constructions the term t may contain literals and the function *if* , thus we will use the algorithm A_F to "flatten out", so to say, any literals containing the term t after direct substitution.

Definition *Let F be a formula with the main variable y and let S be the clause form of F. Let t be a ground B-correct term, possibly containing literals and the function if .*

Consider the clause set S to be a conjunction of disjunctions of literals. Build a new construction S_t by replacing the variable y everywhere in S by the term t. Build the formula $S_{A_F(t)}$ by replacing all the literals L in S_t containing t by the formula computed by $A_F(L)$. $S[t/y]$ is obtained by bringing the formula $S_{A_F(t)}$ to the conjunctive normal form and removing all the tautologous clauses from the result.

In order to define $F[t/y]$ for the formula F, possibly containing quantifiers, we introduce some extra constructions to overcome the problems with the parametric Skolem constants in S and t. Let $c_1, \ldots, c_n$ be the parametric Skolem constants in S replacing the parametric variables $x_1, \ldots, x_n$ in F. Construct the term t' from t by replacing the constants $c_1, \ldots, c_n$ in t by the corresponding variables $x_1, \ldots, x_n$ in F. Build a new construction F'_t by removing the quantifier binding the variable y and replacing y everywhere in F by the term t'. $F[t/y]$ is obtained from F'_t by replacing all the literals L in F_t containing t' by the formula computed by $A_F(L)$.

Given a formula F, its clause form S and a ground term t not containing any nonparametric Skolem functions in S, it is easy to see that $S[t/y]$ is unsatisfiable iff $F[t/y]$ is provable.

The following example illustrates the construction of $S[t/y]$.

Example Let $F \doteq ((P(a) \vee P(b)) \Rightarrow \exists y P(y))$. The clause form S:

$$S \doteq \{\{P(a), P(b)\}, \{\neg P(y)\}\}$$

Let t be the term $if(P(b), b, a)$. Then $S[t/y]$ is:

$$\{\{P(a), P(b)\}, \{\neg P(b)\}, \{P(b), \neg P(a)\}\}$$

which is obtained by converting the following formula to the conjunctive normal form:

$$(P(a) \vee P(b)) \,\&\, ((P(b) \Rightarrow \neg P(b)) \,\&\, (\neg P(b) \Rightarrow \neg P(a)))$$

which is obtained from the following construction by one application of the algorithm A_F:

$$(P(a) \vee P(b)) \,\&\, (\neg(P(if(P(b), b, a))))$$

The following theorem shows that for the Simple Class the A(R)-calculus is complete even in the standard classical sense.

Theorem 1 *Let F be a formula in the Simple Class and let the term restriction $R(t)$ return True for every term t. Let S be a clause form of F with the answer literals $A(y)$ added.*

If F is classically provable, then the A(R)-calculus derives from S a definite answer clause which is either empty or has a form $A(g)$ such that $F[g/y]$ is provable.

Proof For any clause $\{A(t_1), \ldots, A(t_n), B_1, \ldots, B_m\}$ containing several occurrences of the answer predicate and derivable by the standard resolution calculus, there is a clause $\{A(t), B_1, \ldots, B_m\}$ derivable by the A(R)-calculus. As S does not contain nonparametric Skolem functions, g does not contain them either. Finally, use the fact that the A_F algorithm performs only equivalent transformations and use the correctness lemma of the ANS-method from [1]. □

The following completeness theorem for the general case is a main result of the paper. The idea of the proof is similar to the idea of the earlier D-completeness proof.

Theorem 2 *The A(R)-calculus is $D_{if}(R)$-complete.*

We will present a short summary of the structure of the proof. The proof can be considered to consist of two parts. The first part, before the definition of "final clauses", introduces several constructions necessary for the main part of the proof starting after the mentioned definition.

We consider a clause set S and a B-correct ground term t such that $S\{t/y\}$ is unsatisfiable. Thus also the clause set $S[t/y]$ is unsatisfiable.

We take an arbitrary unsatisfiable Herbrand expansion of $S[t/y]$ and we use standard ordered resolution on this expansion, since we already know that it is complete. The clause set $S[t/y]_{EA}$ is obtained by adding answer literals to the expansion. We introduce a class of clauses called "final". After the last definition most of the necessary constructions have been built and the main part of the proof starts.

We show that it is impossible for the standard ordered resolution to infer a non-final clause when at least one of the premisses is final. After that we show that from two nonfinal clauses the standard ordered resolution cannot derive a nontautologous clause containing several different answer-literals. Any standard ordered derivation from the final clauses translates directly into a successful A-resolution derivation. Finally, observe that all the clauses in the set $S[t/y]_{EA}$ are subsumed by the set S modified by adding answer literals. Use lifting, which is possible due to the ordered form of the derivation from $S[t/y]_{EA}$.

Proof Recall the definition of $D_{if}(R)$-completeness: we assume that we have a clause set S and there is a B-correct ground term t such that $S\{t/y\}$ is unsatisfiable. In that case also the clause set $S[t/y]$ is unsatisfiable.

Recall the construction of $S[t/y]$. We denote the set of literals in $S[t/y]$ built from the literals in the term t by the algorithm A_F as I. Notice that since t is ground, all the literals in I are also ground. Further, each literal in I has both a positive and a negative occurrence in I.

Let $G \doteq (g_1, \ldots, g_l)$ be the sequence of all non-*if*-terms occurring in the treeyified form of t as the second and third arguments of the function *if*. G represents the possible choices given by t for the term to be substituted, so to say. For each element g_i in G there is a corresponding choice path in the treeyified form of the term t. Let Δ_i be the set of all the literals on that path, but in the negated form (the element g_i is chosen by t iff all the literals in Δ_i have a truth value *False*). We call the set Δ_i a *path-clause* of the term g_i. Due to the construction of the term t, the set of the path-clauses of all elements of G is unsatisfiable. Further, for any two different path-clauses Δ_i and Δ_j $(i \neq j)$ we know that the clause $\Delta_i \cup \Delta_j$ is a tautology. Notice also that each literal I has both a positive and a negative occurrence in I.

Let S_y be the set of those clauses in S which contain the variable y and let S_o be the set of all the other clauses in S. Notice that in $S[t/y]$ all the clauses in S_o are preserved unchanged. $S[t/y] = S_o \cup S_t$, where S_t is built from the clauses $S_y \doteq \{C_1, \ldots, C_k\}$ and can be assumed to have a following form:

$$\{C_1\{g_1/y\} \cup \Delta_1, \ldots, C_1\{g_l/y\} \cup \Delta_l, \ldots, C_k\{g_1/y\} \cup \Delta_1, \ldots, C_k\{g_l/y\} \cup \Delta_l\}$$

(with the tautologous element clauses missing due to tautologies being removed by the construction of $S[t/y]$).

Since $S[t/y]$ is unsatisfiable, there must be some unsatisfiable finite Herbrand expansion $S[t/y]_E$ of the set $S[t/y]$. Recall that the finite Herbrand expansion of some clause set $\{C_1, \ldots, C_n\}$ is the set

$$\{C_1\sigma_1, \ldots, C_1\sigma_m, \ldots, C_n\sigma_1, \ldots, C_n\sigma_m\}$$

where each $C_i\sigma_j$ is ground and contains only predicate, function and constant symbols from the set $\{C_1, \ldots, C_n\}$ (plus a single new constant symbol, in case $\{C_1, \ldots, C_n\}$ contains none).

Unless it is explicitly said otherwise, we will in the following use ordinary resolution (not the D- or A-calculus) which is restricted in the following completeness-preserving manner. We introduce a following ordering of ground literals in $S[t/y]_E$: all the literals in $S[t/y]_E$ which do not occur in the set I are preferred for resolution over the literals occurring in I. We restrict the resolution method by allowing resolution upon a literal L in a clause C only if C does not contain any literal R preferred over L. This restriction is a case of so-called ordered semantic resolution, see [3] or [1]. We will restrict resolution further by prohibiting the derivation of tautologies (clauses containing some literal L and its negation $\neg L$). This restriction preserves completeness for the semantic resolution.

We build the clause set $S[t/y]_{EA}$ from the clause set $S[t/y]_E$ by adding an answer literal $\{A(g_i)\}$ to each clause $C_j\{g_i/y\} \cup \Delta_i$ $(1 \leq j \leq k, 1 \leq i \leq l)$ built from some clause C_j in S_y.

Definition *A clause in $S[t/y]_{EA}$ or derived from $S[t/y]_{EA}$ is called a* final clause *iff it contains only answer literals and literals from I.*

We will first show that if some final clause C is derived from $S[t/y]_{EA}$, it is impossible to use C for the derivation of a nonfinal clause. Consider a resolution inference with premisses being C and some other clause C'. The consequent is non-final only if C' contains literals not occurring in I. Let N be the set of all these literals in C' which do not occur in I.

Due to the construction of N and I, none of the literals in N occur in C neither positively nor negatively. Thus the inference step is possible only if C' contains also some literals from I. But these literals cannot be resolved upon in C' due to the ordering restriction we use.

As a second step we will show that from two non-final clauses C and C' it is impossible to derive a clause C'' such that C'' contains more than one occurrence of an answer literal. We assume that C contains some answer literal $\{A(g_i)\}$ amd C' contains some answer literal $\{A(g_j)\}$. Consider the case $i \neq j$. Since C and C' as non-final clauses cannot have been inferred from final clauses, the consequent of the inference would be a tautology (due to the construction of $S[t/y]_{EA}$) and thus is not allowed to be inferred according to our resolution strategy. Consider the case where $i = j$. Then $\{A(g_i)\}$ is the same as $\{A(g_j)\}$. As they are both ground, the consequent of the inference contains a single answer literal.

As a third step, notice that any clause inferred from two final clauses by an A-resolution inference step contains either no answer literals or a single ground answer literal $A(d)$ where d is constructed from the terms in G and literals in I and the function *if*. Since R is assumed to be strongly liftable, $R(t)$ is assumed to hold and the clause set $S[t/y]_E$ is assumed to be unsatisfiable, the A(R)-calculus derives from $S[t/y]_{EA}$ a definite answer clause with a term satisfying R. Let D be such a derivation.

Finally, consider the original clause set S. Add an answer literal $A(y)$ to each clause containing the variable y. We get the following clause set S':

$$S' \doteq (S_o \cup \{\{C_1, A(y)\}, \ldots, \{C_k, A(y)\}\})$$

For each clause C_i in the set $S[t/y]_{EA}$ there is a clause C'_j in the set S' subsuming C_i (in general, several clauses in $S[t/y]_{EA}$ may map to one clause in S').

We will now lift the derivation D from the clause set $S[t/y]_{EA}$ to the derivation of a definite answer-clause from the set S'. We note that the standard lifting lemma is not true for the A(R)-calculus due to the D-resolution rule. However, we can show that D can be assumed to have a special form such that the standard lifting lemma is applicable. Namely, whenever there is a derivation of a definite answer-clause from the final clauses, then there is also a derivation without the use of the D-resolution rule (since literals in final clauses satisfy the R-restriction, D-resolution inferences can be replaced by A-resolution inferences). Considering D-resolution inferences from the non-final clauses, we observe that the answer literals in the figure do not contain *if* and thus standard lifting is applicable.

Lifting: transform the derivation D to a derivation D' by replacing each input clause C_i in $S[t/y]_{EA}$ by the subsuming clause C'_j in S' and each clause inferred in D by the correspondingly inferred subsuming clause. Remove the resolution inferences which become impossible (it is possible to remove those in lifting since for such figures the lifted consequent is the same as the lifted premiss).

Conclusion: since $S[t/y]$ is assumed to be provable, $S[t/y]_E$ is an arbitary finite unsatisfiable Herbrand expansion of $S[t/y]$, the term with the required properties was derivable from the clause set $S[t/y]_{EA}$ by the ordered A(R)-resolution and the term restriction R is strongly liftable, the term with the required properties is also derivable from the clause set S' by the unrestricted A(R)-resolution. □

By examining the proofs of the completeness theorems in the paper we can easily see that the results hold also in case full subsumption and tautology elimination are used during proof search. We will present a lemma guaranteeing $D_{if}(R)$-completeness of a subset of ordering restrictions of resolution which preserve (standard) completeness of resolution. We say that an ordering $\succ$ of literals preserves (standard) completeness of resolution iff for any unsatisfiable clause set S there is a derivation of an empty clause such that a literal L in a clause C is not resolved upon in case there is a literal L' in C such that $L' \succ L$. See [3] for the detailed analysis of ordering restrictions.

Lemma 3 *Let $\succ$ be an ordering of literals and let $\succ$ preserve completeness of resolution. Then $\succ$ will preserve $D_{if}(R)$-completeness of A(R)-resolution if both of the following hold:*

- *liftability:* $\forall L, L', \sigma.(L' \succ L) \Rightarrow (L'\sigma \succ L\sigma)$
- *R-compatibility:* $\forall L.\neg R(L) \Rightarrow \neg\exists L'(L' \succ L)$

The proof can be inferred by examining the proof of the main theorem above. We remark that the given criteria can be strengthened by comparing literals in the scope of a derivable clause instead of comparing all the possible literals.

The following is an example of $D_{if}(R)$-completeness being lost due to the ordering not satisfying R-compatibility. We remark that since hyperresolution is essentially an optimised version of a certain kind of ordered binary resolution, hyperresolution is not $D_{if}(R)$-complete in the general case. Compatibility of a set of hyperresolution nucleons with a given restriction R has to be checked for each particular case.

Example Let $\succ_G$ be defined as "literals with the predicate G are preferred over all the other literals". The ordering $\succ_G$ is an instance of the semantic resolution and is thus known to preserve completeness of resolution. Consider the clause set S:

$\{\neg G(b,y),\ P(a),\ A(y)\}$

$\{G(y,a),\ P(b),\ A(y)\}$

$\{\neg P(y),\ A(y)\}$

We define $R(t)$ as "t does not contain the predicate symbol P". Then there is no A(R)-derivation of a definite answer clause from S satisfying the ordering $\succ_G$, whereas there is an A(R)-derivation of the definite answer clause $\{A(if(G(b,a),a,b))\}$ from S using unrestricted A(R)-calculus. This derivation relies on the D-resolution rule.

5 Examples of program synthesis

We will present four examples of program synthesis using the A(R)-algorithm. We will use the paramodulation rule instead of an explicit axiomatization of equality:

$$\frac{\{L[t],\Gamma\}\quad\{t'=g,\Delta\}}{\{L[g],\Gamma,\Delta\}\sigma}\sigma = mgu(t,t')$$

where $L[g]$ is obtained by replacing one occurrence of the term t in $L[t]$ by the term g. The equality predicate in the rule is assumed to be commutative, i.e. $t'=g$ is the same as $g=t'$. The terms t and t' must be unifiable.

The completeness theorem for paramodulation (see [12]) says that if a clause set $S \cup E$ is unsatisfiable, where E is an axiomatization of the equality predicate $=$ for S, then the empty clause will be derived from the set $S \cup \{\{x=x\}\}$ by resolution, factorization and paramodulation steps. Completeness is preserved if the term t in the paramodulation rule is prohibited to be a variable.

Since each inference by the paramodulation rule can be replaced by a figure containing only the premisses of paramodulation, resolution rule applications and axioms of equality, the A(R)-calculus is easily extended to paramodulation. We will skip the details here. We will call the axiom $x=x$ as "$=$ reflexivity" and we will not list it explicitly among the axioms.

The programs are always assumed to take only lists as inputs and in the following formalizations the explicit type information is skipped.

Definition Lists *are built inductively from the constant nil (empty list) and arbitrary objects (list members) by the pair constructor c, so that a list $c(h, x)$ is obtained from the list x by prepending a new element h.*

We take a predicate m such that $m(x, y)$ means: x is a member of the list y. We take as an axiom the following formula defining m:

$$(\forall x(\neg m(x, nil))) \,\&\, (\forall x, y, z(m(x, c(y, z)) \Leftrightarrow (x = y \vee m(x, z))))$$

In the following we assume that m is decidable, i.e. always define the term restriction $R(t)$ so as to allow t to contain m.

We will use structural induction over lists:

$$\frac{\forall x_2 \ldots x_n \exists y.A\{nil/x_1\} \qquad \forall x((\forall x_2 \ldots x_n \exists y_1.A\{x/x_1, y_1/y\}) \Rightarrow (\forall h x_2 \ldots x_n \exists y.A\{c(h, x)/x_1\}))}{\forall x_1 \ldots x_n \exists y.A}$$

In the second premiss of the scheme above the variable y_1 is bound by the essentially universal quantifier and has an interpretation as a recursive case of the program for computing y. We will present the program extracted from the proof of the basis and step formulas (see [11]) as two equalities, one for the constructor *nil* and one for the pair constructor c.

The choice of the induction principle is not relevant for our aim of demonstrating the A(R)-calculus proofs of the first-order tasks. We have picked the structural induction principle (although there exist much stronger induction principles) solely for the reasons of simplicity of presentation. The examples we have chosen also do not need additional lemmas, although in the general case lemmas may be needed for proofs using induction. The lack or presence of additional lemmas is irrelevant for the A(R)-calculus.

5.1 Checking whether a list contains a certain element

Take the previous definition of a decidable list membership predicate m.

Derive a program to check that a given list contains an element 0 by returning a value 1 if and only if the list contains 0:

$$\forall x(\exists y(m(0, x) \Leftrightarrow y = 1))$$

and we want to find a program to compute a value of y for any list-type value of x. We need a constant which would be provably unequal to 1. We take 0 to be unequal to 1 by just assuming $0 \neq 1$.

Conversion of the whole problem to the resolution form (overlined variables like $\overline{x}$ are Skolem functions, first four clauses come from the definition of m, fifth is the assumed unequality of 0 and 1, A is the answer predicate to collect substitutions, clauses 6 and 7 come from the formula above.

1) $\{\neg m(x, nil)\}$

2) $\{\neg m(x, c(y, z)),\ x = y,\ m(x, z)\}$

3) $\{x \neq y,\ m(x, c(y, z))\}$

4) $\{\neg m(x, z),\ m(x, c(y, z))\}$

5) $\{0 \neq 1\}$

6) $\{m(0, \overline{x}),\ y = 1,\ A(y)\}$

7) $\{\neg m(0, \overline{x}),\ y \neq 1,\ A(y)\}$

The refutation:

7 and $=$ reflexivity give 8) $\{\neg m(0, \overline{x}),\ A(1)\}$

6 and 5 give 9) $\{m(0, \overline{x}),\ A(0)\}$

8 and 9 give contradiction: $\{A(if(m(0, \overline{x}), 1, 0)\}$

The substitution $if(m(0, \overline{x}), 1, 0)$ found is computable, as $if, m, 0$ and 1 are assumed to be computable and $\overline{x}$ has an interpretation as an input list. The following is the deduced program:

$$y(x) \quad = \quad if(m(0, x), 1, 0)$$

5.2 Another algorithm for checking whether a list contains a certain element

In this example we take a list membership predicate m' defined exactly in the same way as the predicate m, but lacking the decidability proof: we *do not assume* that the predicate m' is decidable.

We pose the same problem as in the last example, with a single difference that m' is used instead of m. The proof from the last example fails, as the substitution $if(m'(0, \overline{x}), 1, 0)$ cannot be assumed to be computable, since m' is not assumed to be computable.

We get a successful proof by using one structural induction over x:

5.2.1 Induction base

$$\exists y(m'(y, nil) \Leftrightarrow y = 1)$$

Conversion of the problem to the resolution form:

1) $\{\neg m'(x, nil)\}$

2) $\{\neg m'(x, c(y, z)),\ x = y,\ m'(x, z)\}$

3) $\{x \neq y,\ m'(x, c(y, z))\}$

4) $\{\neg m'(x, z),\ m'(x, c(y, z))\}$

5) $\{0 \neq 1\}$

6) $\{m'(0, nil),\ y = 1,\ A(y)\}$

7) $\{\neg m'(0, nil),\ y \neq 1,\ A(y)\}$

The refutation:

6,1 and 5 give 8) $\{A(0)\}$

Thus the base case for the program is

$$y(nil) = 0$$

5.2.2 Induction step

$$\forall x((\exists y(m'(0, x) \Leftrightarrow y = 1)) \Rightarrow (\forall v(\exists u(m'(0, c(v, x)) \Leftrightarrow u = 1))))$$

Conversion of the problem to the resolution form:

1) $\{\neg m'(x, nil)\}$

2) $\{\neg m'(x, c(y, z)),\ x = y,\ m'(x, z)\}$

3) $\{x \neq y,\ m'(x, c(y, z))\}$

4) $\{\neg m'(x, z),\ m'(x, c(y, z))\}$

5) $\{0 \neq 1\}$

6) $\{\neg m'(0, \overline{x}),\ \overline{y} = 1\}$

7) $\{m'(0, \overline{x}),\ \overline{y} \neq 1\}$

8) $\{m'(0, c(\overline{v}, \overline{x})),\ u = 1,\ A(u)\}$

9) $\{\neg m'(0, c(\overline{v}, \overline{x})),\ u \neq 1,\ A(u)\}$

The refutation:

6,2 give 10) $\{\overline{y} = 1,\ \neg m'(0, c(y, \overline{x})),\ 0 = y\}$

7,4 give 11) $\{\overline{y} \neq 1,\ m'(0, c(y, \overline{x}))\}$

10,9 give 12) $\{\neg m'(0, c(y, \overline{x})),\ 0 = y,\ \neg m'(0, c(\overline{v}, \overline{x})),\ A(\overline{y})\}$

12 gives 13) $\{o = \overline{v},\ \neg m'(0, c(\overline{v}, \overline{x})),\ A(\overline{y})\}$

11,8 give 14) $\{m'(0, c(y, \overline{x})),\ A(\overline{y}) m'(0, c(\overline{v}, \overline{x}))\}$

14,13 give 15) $\{m'(0, c(y, \overline{x})),\ 0 = \overline{v},\ A(\overline{y})\}$

15,13 give 16) $\{0 = \overline{v},\ A(\overline{y})\}$

9, = reflexivity give 17) $\{\neg m'(0, c(\overline{v}, \overline{x})),\ A(1)\}$

17,3 give 18) $\{0 \neq \overline{v},\ A(1)\}$

16,18 give contradiction: $\{A(if(\overline{v} = 0, 1, \overline{y}))\}$

As the Skolem function $\overline{v}$ has an interpretation as an argument of the program for u and the function $\overline{y}$ denotes the program for the induction hypothesis, the term found is computable and the inductive case for the program is:

$$u(c(v,x)) = if(v = 0, 1, u(x))$$

As a whole the program for computing y is:

$$\begin{aligned} y(nil) &= 0 \\ y(c(v,x)) &= if(v = 0, 1, y(x)) \end{aligned}$$

5.3 Finding a certain element of the list

Take the previous definition of a list membership predicate m and assume m to be decidable. Take an arbitrary decidable predicate P. Derive a program to find a member of a list satisfying P, under assumption that the list contains such a member. The specification is

$$\forall x((\exists y(m(y,x)\,\&\,P(y))) \Rightarrow (\exists z(m(z,x)\,\&\,P(z))))$$

and we want to find a program to compute a value for z for any list-type value of x. We define $R(t)$ as "t does not contain the Skolem function for y".

First, an attempt to derive a definite answer clause with a term satisfying R fails, if we do not use induction. Conversion of the whole problem to the resolution form (overlined variables like $\overline{x}$ represent Skolem functions, first four clauses come from the definition of m, A is the answer predicate to collect substitutions):

1) $\{\neg m(x, nil)\}$

2) $\{\neg m(x, c(y,z)),\ x = y,\ m(x,z)\}$

3) $\{x \neq y,\ m(x, c(y,z))\}$

4) $\{\neg m(x,z),\ m(x, c(y,z))\}$

5) $\{m(\overline{y}, \overline{x})\}$

6) $\{P(\overline{y})\}$

7) $\{\neg m(z, \overline{x}),\ \neg P(z),\ A(z)\}$

The only derivable definite answer clause is derived in the following way:

5,6 and 7 give 8) $\{A(\overline{y})\}$

But this answer is discarded by R.

We get successful derivations of a definite answer clause by using one structural induction over x:

5.3.1 Induction base

$$(\exists y(m(y, nil)\,\&\,P(y))) \Rightarrow (\exists z(m(z, nil)\,\&\,P(z)))$$

Conversion of the problem to the resolution form:

1) $\{\neg m(x, nil)\}$

2) $\{\neg m(x, c(y, z)),\ x = y,\ m(x, z)\}$

3) $\{x \neq y,\ m(x, c(y, z))\}$

4) $\{\neg m(x, z),\ m(x, c(y, z))\}$

5) $\{m(\overline{y}, nil)\}$

6) $\{P(\overline{y})\}$

7) $\{\neg m(z, nil),\ \neg P(z),\ A(z)\}$

There is a one-step refutation: clauses 1 and 5 give contradiction. This refutation does not instantiate the variable z. Thus any substitution to variable z is admissible, and the base case for the program is

$$z(nil) = t$$

where t is an arbitrary object. Indeed, due to the assumption the base case will be never reached, thus the value of z on nil does not have any importance.

5.3.2 Induction step

$$\begin{aligned}&\forall x(((\exists y(m(y, x)\,\&\,P(y))) \Rightarrow (\exists z(m(z, x)\,\&\,P(z)))) \Rightarrow \\ &\quad \forall w((\exists u(m(u, c(w, x))\,\&\,P(u))) \Rightarrow (\exists v(m(v, c(w, x))\,\&\,P(v))))))\end{aligned}$$

and we want to get a program to compute the value of v for any value of w (head of the list) and x (tail of the list).

The following variables have interpretation: w and x as formal parameters, z as an induction hypothesis (the program for the recursive case). We define $R(t)$ as "t does not contain Skolem functions for y and u".

The resolution form:

1) $\{\neg m(x, nil)\}$

2) $\{\neg m(x, c(y, z)),\ x = y,\ m(x, z)\}$

3) $\{x \neq y,\ m(x, c(y, z))\}$

4) $\{\neg m(x, z),\ m(x, c(y, z))\}$

5) $\{\neg m(y, \overline{x}),\ \neg P(y),\ m(\overline{z}, \overline{x})\}$

6) $\{\neg m(y, \overline{x}),\ \neg P(y),\ P(\overline{z})\}$

7) $\{m(\overline{u}, c(\overline{w}, \overline{x}))\}$

8) $\{P(\overline{u})\}$

9) $\{\neg m(v, c(\overline{w}, \overline{x}),\ \neg P(v),\ A(v)\}$

The derivation:

2,7 give 10) $\{\overline{u} = \overline{w},\ m(\overline{u}, \overline{x})\}$

9,3 give 11) $\{v \neq \overline{w},\ \neg P(v),\ A(v)\}$

9,4 give 12) $\{\neg m(v, \overline{x}),\ \neg P(v),\ A(v)\}$

12,10 give 13) $\{\overline{u} = \overline{w},\ \neg P(\overline{u}),\ A(\overline{u})\}$

10,5 give 15) $\{\neg P(\overline{u}),\ \overline{u} = \overline{w},\ m(\overline{z}, \overline{x})\}$

15,8 give 16) $\{\overline{u} = \overline{w},\ m(\overline{z}, \overline{x})\}$

10,6 give 17) $\{\neg P(\overline{u}),\ \overline{u} = \overline{w},\ P(\overline{z})\}$

17,8 give 18) $\{\overline{u} = \overline{w},\ P(\overline{z})\}$

16,4 give 19) $\{\overline{u} = \overline{w},\ m(\overline{z}, c(y, \overline{x}))\}$

19,9 give 20) $\{\overline{u} = \overline{w},\ \neg P(\overline{z}),\ A(\overline{z})\}$

20,18 give 21) $\{\overline{u} = \overline{w},\ A(\overline{z})\}$

21,8 give 30) $\{P(\overline{w}),\ A(\overline{z})\}$

11, = reflexivity give 26) $\{\neg P(\overline{w}),\ A(\overline{w})\}$

26,30 give: $\{A(if(P(\overline{w}), \overline{w}, \overline{z}))\}$

The substitution $if(P(\overline{w}), \overline{w}, \overline{z})$ found gives a program for the recursive case:

$$z(c(w, x)) = if(P(w), w, z(x))$$

As a whole the program is the following:

$$\begin{aligned} z(nil) &= t \\ z(c(w, x)) &= if(P(w), w, z(x)) \end{aligned}$$

where t is arbitrary, i.e. the end of the list is known to be never reached.

5.4 Intersection of two sets

Take the previous definition of a decidable list membership predicate m. We use lists to represent sets and will derive a program to compute the intersection of two sets. The specification:

$$\forall x \forall y \exists z \forall u((m(u, x) \,\&\, m(u, y)) \Leftrightarrow m(u, z))$$

and we want to find a program to compute a value for z for any list-type values of x and y.

An attempt to prove the formula above without induction fails even for unrestricted classical logic. We will use structural induction over the first list-type variable x.

5.4.1 Induction base

$$\forall y \exists z \forall u((m(u, nil) \,\&\, m(u, y)) \Leftrightarrow m(u, z))$$

and we define $R(t)$ as "t does not contain the Skolem function for u". Conversion of the whole problem to the resolution form (overlined variables like $\overline{x}$ represent Skolem functions, first four clauses come from the definition of m, A is the answer predicate to collect substitutions):

1) $\{\neg m(x, nil)\}$

2) $\{\neg m(x, c(y, z)),\ x = y,\ m(x, z)\}$

3) $\{x \neq y,\ m(x, c(y, z))\}$

4) $\{\neg m(x, z),\ m(x, c(y, z))\}$

5) $\{m(\overline{u}(z), nil),\ m(\overline{u}(z), z),\ A(z)\}$

6) $\{m(\overline{u}(z), \overline{y}),\ m(\overline{u}(z), z),\ A(z)\}$

7) $\{\neg m(\overline{u}(z), nil),\ \neg m(\overline{u}(z), \overline{y}),\ \neg m(\overline{u}(z), z),\ A(z)\}$

The derivation of the definite answer clause is obtained in two steps from clauses 1 and 5. The program synthesized for the base case:

$$z(nil, y) = nil$$

5.4.2 Induction step

$$\begin{aligned}&\forall x((\forall y \exists z \forall u((m(u, x) \,\&\, m(u, y)) \Leftrightarrow m(u, z))) \Rightarrow \\ &\quad (\forall h \forall y_1 \exists z_1 \forall u_1((m(u_1, c(h, x)) \,\&\, m(u_1, y_1)) \Leftrightarrow m(u_1, z_1))))\end{aligned}$$

and we want to get a program to compute the value of z_1 for any values of h (head of the list) and x (tail of the list).

The following Skolem functions have interpretation: $\overline{h}$ and $\overline{x}$ as formal parameters, $\overline{z}$ as an induction hypothesis (the program for the recursive case). Thus $R(t)$ is defined as "t does not contain the Skolem functions $\overline{y_1}$ and $\overline{u_1}$".

The resolution form:

1) $\{\neg m(x, nil)\}$

2) $\{\neg m(x, c(y, z)),\ x = y,\ m(x, z)\}$

3) $\{x \neq y,\ m(x, c(y, z))\}$

4) $\{\neg m(x, z),\ m(x, c(y, z))\}$

5) $\{\neg m(u, \overline{x}),\ \neg m(u, y),\ m(u, \overline{z}(y))\}$

6) $\{m(u, \overline{x}),\ \neg m(u, \overline{z}(y))\}$

7) $\{m(u, y),\ \neg m(u, \overline{z}(y))\}$

8) $\{m(\overline{u_1}(z_1), c(\overline{h}, \overline{x})),\ m(\overline{u_1}(z_1), z_1),\ A(z_1)\}$

9) $\{m(\overline{u_1}(z_1), \overline{y_1}),\ m(\overline{u_1}(z_1), z_1),\ A(z_1)\}$

10) $\{\neg m(\overline{u_1}(z_1), c(\overline{h}, \overline{x})),\ \neg m(\overline{u_1}(z_1), \overline{y_1}),\ \neg m(\overline{u_1}(z_1), z_1),\ A(z_1)\}$

The shortest derivation of the definite answer that we have found has depth 10 and contains 27 hyperresolution, paramodulation and factorization steps. We'll skip the proof here. The program synthesized for the inductive case:

$$z_1(c(h,x),y) = if(m(h,y), c(h, z_1(x,y)), z_1(x,y))$$

As a whole the program is the following:

$$\begin{aligned} z(nil,y) &= nil \\ z(c(h,x),y) &= if(m(h,y), c(h, z(x,y)), z(x,y)) \end{aligned}$$

6 Summary

We consider the resolution-based A(R)-calculus for deriving conditional programs from classical first order specifications. This calculus is essentially the same as the A-resolution method introduced in [2] and [1]. Moreover, the A(R)-calculus corresponds to the clause fragment of the calculus used by Z.Manna and R.Waldinger, see [6] and [7] (Z.Manna and R.Waldinger use full first-order language).

We formulate and prove completeness of the A(R)-calculus, giving also a completeness-preserving criteria (strong liftability) for the special term restriction R necessary for program derivation. We give a completeness-preserving criteria and a counterexample for ordering restrictions of A(R)-calculus. The earlier completeness result by U.Schmerl in [14] for the resolution-based calculus of deriving definite answers without conditions (ie. without *if*) and without the R-restriction is a subcase of our result.

Acknowledgements: We'd like to thank Jan Smith, Tarmo Uustalu and especially Grigori Mints for useful discussions, criticism, numerous ideas and suggestions. We are also grateful to several anonymous referees.

References

[1] C.L.Chang, R.C.T Lee. Symbolic Logic and Mechanical Theorem Proving. *Academic Press*, 1973.

[2] C.L.Chang, R.C.T Lee, R.Waldinger. An Improved Program-Synthesizing Algorithm and its Correctness. *Comm. of ACM*, (1974), V17, N4, 211-217.

[3] C.Fermüller, A.Leitsch, T.Tammet, N.Zamov. Resolution Methods for the Decision Problem. *Lecture Notes in Artificial Intelligence 679,* Springer Verlag, 1993.

[4] C.Green. Application of theorem-proving to problem solving. In *Proc. 1st Internat. Joint. Conf. Artificial Intelligence*, pages 219-239, 1969.

[5] S.C.Kleene. Introduction to Metamathematics. *North-Holland, Amsterdam,* 1952.

[6] Z.Manna, R.Waldinger. A deductive approach to program synthesis. *ACM Trans. Programming Languages and Systems*, (1980), N2(1), 91-121.

[7] Z.Manna, R.Waldinger. Fundamentals of Deductive Program Synthesis. *IEEE Transactions on Software Engineering*, (1992), V18, N8, 674-704.

[8] G.Mints, E.Tyugu. Justification of the structural synthesis of programs. *Sci. of Comput. Program.*, (1982),N2, 215-240.

[9] G.Mints. Gentzen-type Systems and Resolution Rules. Part I. Propositional Logic. In *COLOG-88*, pages 198-231. *Lecture Notes in Computer Science vol. 417, Springer Verlag,* 1990.

[10] G.Mints. Gentzen-type Systems and Resolution Rules. Part II. Predicate Logic. In *Logic Colloquium '90.*

[11] B.Nordström, K.Petersson, J.M.Smith. Programming in Martin-Löf's Type Theory. *Clarendon Press*, Oxford, 1990.

[12] G.Peterson. A technique for establishing completeness results in theorem proving with equality. *SIAM J. of Comput.* (1983), N12, 82-100.

[13] J.A. Robinson. A Machine-oriented Logic Based on the Resolution Principle. *Journal of the ACM* 12, 1965, pp 23-41.

[14] U.R.Schmerl. A Resolution Calculus Giving Definite Answers. *Report Nr 9108, July 1991, Fakultät für Informatik, Universität der Bundeswehr München.*

[15] N.Shankar. Proof Search in the Intuitionistic Sequent Calculus. In *CADE-11*, pages 522-536, *Lecture Notes in Artificial Intelligence 607, Springer Verlag*, 1992.