A Second-order Theorem Prover applied to Circumscription

Michael Beeson

Department of Mathematics and Computer Science
San Jose State University

Abstract. Circumscription is naturally expressed in second-order logic, but previous implementations all work by handling cases that can be reduced to first-order logic. Making use of a new second-order unification algorithm introduced in [2], we show how a theorem prover can be made to find proofs in second-order logic, in particular proofs by circumscription. We work out a blocks-world example in complete detail and give the output of an implementation, demonstrating that it works as claimed.

1 Introduction

Circumscription was introduced by John McCarthy [10] as a means of formalizing "common-sense reasoning" for artificial intelligence. It served as the foundation of his theory of non-monotonic reasoning. The essential idea is to introduce, when axiomatizing a situation, a predicate *ab* for "abnormality", and to axiomatize the *ab* predicate by saying it is the least predicate such that the other axioms are valid. Some other predicates may be allowed to "vary" in the minimization as well. There are several technical difficulties with McCarthy's idea: First, the circumscription principle is most naturally expressed in second-order logic, where we have variables over predicates of objects. Second, unless the rest of the axioms contain *ab* only positively, the circumscription principle is not an ordinary inductive definition, and there may not even be a (unique) least solution for the *ab* predicate, so the circumscription principle can be inconsistent. McCarthy's ultimate goal was implementation of software using the circumscription principle to construct artificial intelligence. Believing that implementation of second-order logic was not a practical approach, many researchers have tried various methods of reducing special cases of the circumscription principle to first-order logic; see [6] for a summary of these efforts. Some of these reductions were in turn implemented.

In this paper we take the other path, and exhibit a direct implementation of second-order logic which is capable of handling some circumscription problems. The key to making this work is a new notion of second-order unification. This notion of unification was introduced in [2], where some theorems about it are proved. In that paper, I pointed out the possibility of converting your favorite first-order theorem prover to a second-order theorem prover by adding second-order unification. This paper shows explicitly how this can be done, and that the resulting second-order prover can indeed find circumscription proofs. Note that it would already be interesting if the resulting proof-checker could accept and verify circumscription proofs, but the essential point of this paper is that the use of the new unification algorithm of [2] enables a simple theorem-prover to find circumscription proofs by itself. The hard part of this, of course, is finding the correct values of the second-order predicates involved. These are generally give by λ terms involving an operator for definition by cases. It is therefore essential to use a formalization of second-order logic which has terms for definition by cases.

Although we have used a specific first-order prover in this exercise, we make no claims about the value either of this particular prover or of the backwards-Gentzen approach that it uses. We believe the same results could be attained with any medium-to-good quality first-order theorem prover, suitably extended to second-order by implementing the new unification algorithm. We just used the theorem-prover we have, in order to demonstrate that this approach works.

On the other hand, it may not be completely trivial to add the new unification algorithm to an existing resolution-style prover such as Otter. Note that in such provers, "variable" means what we here call "metavariable", and our "object variables" are just constants. In such provers, there is no notion of a "restriction" that prevents unification from making the value of a variable depend on certain constants. In a Gentzen-style prover, when we prove $\exists x \forall y P(x,y)$, we try $P(X,y)$ after restricting X to not depend on y. In resolution provers, this is handled by Skolemization, so that when we prove $\exists x \forall y P(x,y)$, we replace y by $g(x)$ and try to refute $\neg P(x, g(x))$. Then the occurs check prevents the eventual value of x from depending on y. The expression $g(x)$ functions essentially as a variable y whose presence causes x to not depend on y. Our new unification algorithm, however, calls for introducing new variables dynamically and restricting their eventual values. Thus it is not just a matter of adding a few lines to Otter's unification code to make this work with Otter. However, it is certainly possible.

2 Definitions

2.1 Syntax of second-order logic

Second-order logic refers to a system in which we have two kinds of variables, object variables and predicate variables. We write lower-case letters $x, y, \ldots$ for object variables and upper-case letters $X, Y, \ldots$ for predicate variables. Similarly, we have object terms and predicate terms. The term formation rules are as follows:

```
Variable:= ObjectVariable | PredicateVariable
BinaryConnective :- ∨ | ∧ | →
ListTerm := [ObjectTerm] | [ObjectTerm | ListTerm]
VarList := [ObjectVariable] | [ObjectVariable | VarList]
ObjectTerm := FirstOrderTerm | ApplicationTerm
ApplicationTerm :=  ap(PredicateTerm, ListTerm)
```

```
PredicateTerm := LambdaTerm | PredicateVariable | PredicateConstant
LambdaTerm := (λ VarList. Formula)
Formula := AtomicFormula | (Formula BinaryConnective Formula) |
(¬ Formula) | CaseTerm | ∀ Variable . Formula
CaseTerm:= d(ObjectTerm , ObjectTerm , PredicateTerm, PredicateTerm)
```

Note that function symbols taking predicate arguments are not legal. There can be function symbols, as usual in first-order logic, but they take only first-order arguments.

As usual, $[x_1, \ldots, x_n]$ abbreviates $[x_1|[x_2, \ldots, x_n]]$. Each `ListTerm` has a unique length n. An `AtomicFormula` is an expression of the form $P(z)$, where P is a predicate term of arity n and z is a list term of length n. In case P is a predicate constant, we consider $P([x_1, \ldots, x_n)])$ to be the same as the usual first-order term $P(x_1, \ldots, x_n)$. For the preceding to make sense, we must define the arity of each predicate term. This is simply the number of its free object variables. In the usual way, we can associate to each predicate or object term a list of its free variables (in a specified order). The syntax rule for `ApplicationTerm` is then subjected to the restriction that $\mathbf{ap}(P, z)$ can be formed only when the length of the list z is equal to the arity of P. variables are bound by the λ operator in the usual way. Thus for example $\lambda x.P(x, y)$ is a predicate term of arity 1. We can therefore form the object term $\mathbf{ap}(\lambda x.P(x, y), c)$.

We include two predicate constants of arity 0, namely **true** and **false**. The above rules then make these two into atomic formulae as well. The use of **true** and **false** is primarily a matter of notational convenience; **false** in the succedent is traditionally written as an empty succedent. We could regard $\neg A$ as an abbreviation for $A \to$ **false**, but it is convenient to retain both notations. However, we allow $\neg A$ and $A \to$ **false** to unify.

$\mathbf{ap}(X, z)$ can be abbreviated in writing as $X(z)$, although in implementations it is maintained, and is printed in the output of our prover. Note, though, that predicate symbols P (constants) of the language are used in the usual first-order syntax $P(x)$, rather than $\mathbf{ap}(P, x)$.

Second-order unification, and its application to circumscription, both depend on the use of conditional terms, or case-terms. These are terms of the form

$$\begin{cases} P(x) & \text{if} \quad x = y \\ Q(x) & \text{ow} \end{cases}$$

There are several different notations for such terms in use, including the form used in the C and Java programming languages:

$$x = y \ ? \ P(x) \ : Q(x)$$

and the form used in [2] and in the theories of Feferman [1]:

$$\mathbf{d}(x, y, P(x), Q(x)).$$

The form with **d** is the one that has been given in the official syntax above, but the other two forms are both more readable. The syntax used by our computer implementation allows a more general kind of case term in which there can be several cases, instead of just one, before the "otherwise" term. For representing such terms the notation with a brace is more readable, so the output of the prover is presented in that notation. For writing papers, the notation with a question mark is more compact and equally readable, so we will use it in the paper.

We note that everything in this paper applies equally well to higher-order logic. This system would allow terms of every finite type. We can then use the device of "currying" to eliminate the need for list terms: instead of $\mathbf{ap}(P,[x,y])$ we would use $\mathbf{ap}(\mathbf{ap}(P,x),y)$. We then have to define the rules for assigning types to terms, and extend the definition of case terms. In case terms, the first two arguments are still restricted to type 0. This is one of the systems that has been used in [2]. (The other is a more general untyped system called λ-D.) Second-order logic is essentially the type-2 subsystem of higher-order logic. Since second-order logic suffices for circumscription, we work in that system.

2.2 Axioms and rules of second-order logic

We use a Gentzen-sequent formulation of second-order logic. We simply take the usual Gentzen rules (e.g. G3 as in [8]) for both predicate and object quantifiers. The G3 rules need to be supplemented with rules corresponding to the formation of λ-terms and $\mathbf{ap}$-terms, as well as with rules corresponding to the introduction of case terms in both antecedent and succedent. We do not repeat the G3 rules here, but here are the other rules:

$$\frac{t = s, A \Rightarrow C, \Gamma \qquad t \neq s, B, \Gamma \Rightarrow C}{\mathbf{d}(t,s,A,B)), \Gamma \Rightarrow C}$$

$$\frac{t = s \Rightarrow A}{\Gamma \Rightarrow \mathbf{d}(t,s,A,B))}$$
$$\frac{t \neq s \Rightarrow B}{\Gamma \Rightarrow \mathbf{d}(t,s,A,B))}$$

$$\frac{\Gamma \Rightarrow A[t/x]}{\Gamma \Rightarrow (\lambda x.A)t}$$

$$\frac{\Gamma, A[t/x] \Rightarrow \phi}{\Gamma, (\lambda x.A)t \Rightarrow \phi}$$

These last rules allow us to rewrite terms by beta-reduction when searching in "backwards-Gentzen" style for a proof, both in the "assumptions" (antecedent) and the "goal" (consequent). With regard to equality reasoning, we can either include Gentzen-style equality rules, or we can simply regard the equality axioms as part of the axioms Γ of the theory in question. In our implementation, there are certain methods that find proofs involving equality which could be interpreted in either system, but actually do not follow either one very closely. As is well-known, equality reasoning offers difficulties for automated deduction, but these difficulties are not directly relevant to the topics discussed in this paper, except of course that the first-order aspects of the prover must be good enough to deal with the equality reasoning required in the circumscription examples.

2.3 Circumscription

If U and V are predicate expressions of the same arity, then $U \leq V$ stands for $\forall x(U(x) \rightarrow V(x))$. If $U = U_1, \ldots, U_n$ and $V = V_1, \ldots, V_n$ are similar tuples of predicate expressions, i.e. U_i and V_i are of the same arity, $1 \leq i \leq n$, then $U \leq V$ is an abbreviation for $\wedge_{i=0}^{n} U_i \leq V_i$. We write $U = V$ for $U \leq V \wedge V \leq U$, and $U < V$ for $U \leq V \wedge \neg V \leq U$.

Definition 1 (Second-Order Circumscription). *Let P be a tuple of distinct predicate constants, S be a tuple of distinct function and/or predicate constants disjoint from P, and let $T(P; S)$ be a sentence. The second-order circumscription of P in $T(P; S)$ with variable S, written $Circ(T; P; S)$, is given in [6] as*

$$T(P; S) \wedge \forall \Phi \Psi \neg [T(\Phi, \Psi) \wedge \Psi < P]$$

where Φ and Ψ are tuples of variables similar to P and S, respectively. This can equivalently be stated in the form

$$T(P; S) \wedge \forall \Phi \Psi [T(\Phi, \Psi) \wedge \Psi \leq P \rightarrow P \leq \Psi],$$

which is the form our prover uses.

3 Unification

In this section, we recall the notion of unification introduced in [2].

3.1 Metavariables

A *metavariable* is a variable (not part of the formal language) ranging over terms of the formal language. Metavariables are used in a theorem prover to stand temporarily for terms whose values will eventually be determined. Unification is the means by which the values are determined. For example, when the prover tries to prove $\exists Y A(Y)$, a new metavariable $\mathbf{Z}$ is introduced and the goal becomes $A(\mathbf{Z})$. We use boldface letters for metavariables, since the distinction between lower and upper case is already used for something else. In comparing [2] with this paper, it should be understood that the "variables" of [2] are the metavariables of this paper, and the "constants" of [2] are the object and predicate variables (and the constants) of second-order logic.

3.2 Restrictions and Environments

A *restriction* is a pair consisting of a metavariable and a (possibly empty) list of (object or predicate) variables. (Intuitively, the eventual value of the variable is not allowed to depend on the members of the list.) An *environment* is a finite list of restrictions. (Intuitively, an environment lists all the variables in use so far, whether or not their eventual values are restricted, together with any restrictions so far imposed.) If $\langle \mathbf{Z}, r\rangle$ is a member of the environment E we say that the variable $\mathbf{Z}$ *occurs in* E or *is mentioned in* E, and that all the members of the list r are *forbidden to* $\mathbf{Z}$ *in* E. We say a compound term t is *forbidden to* $\mathbf{Z}$ *in* E if it contains a free occurrence of any constant that is forbidden to $\mathbf{Z}$ in E. A *substitution* is a function from metavariables to terms. The substitution σ is *legal for environment* E provided $\sigma(\mathbf{Z})$ is defined for all $\mathbf{Z}$ that occur in E and that $\sigma(\mathbf{Z})$ does not contain free occurrences of any variable or constant forbidden to $\mathbf{Z}$ in E. The substitution σ *unifies* terms t and s *relative to* E if for some substitution χ whose restriction to E is the identity, we have $t\sigma\chi = s\sigma\chi$.[1]

3.3 Definition of unification

The inputs to the unification algorithm are two terms t and s to be unified and an environment E. We say that t and s are to be unified "relative to" the environment E. One output of the unification algorithm is a substitution σ which is legal for E, such that $t\sigma = s\sigma$. The usual notion of unification is obtained by taking an environment E with no restrictions on any of the variables occurring in E. But note that the use of restrictions, even in first-order unification, corresponds to the actual use of unification in theorem-proving, where for example when we try to prove $\Gamma \Rightarrow \exists Y A$, we introduce a new metavariable $\mathbf{Z}$ with the restrictions that its ultimate value cannot depend on variables bound in A, Γ.

The unification algorithm has a second output, which is a new environment (possibly) enlarging the input environment E. Here "enlarging" means simply that new variables may have been added.

The key new clauses in the definition of unification are these:

To unify $\mathbf{Z}(t)$ and S, where t and S are not forbidden to $\mathbf{Z}$, we take $\mathbf{Z} = \lambda x \mathbf{d}(x, t, S, \mathbf{Z}x)$. Here $\mathbf{Z}$ is a new metavariable. The output environment includes Y. The variables forbidden to $\mathbf{Z}$ are the ones forbidden to $\mathbf{Z}$.

To unify $X(t_1, t_2)$ and S, where t_1, t_2 and S are not forbidden to X, we take

$$X = \lambda x_1 x_2 (\mathbf{d}(x_1, t_1, \mathbf{d}(x_2, t_2, s, \mathbf{Z}_2 x), \mathbf{Z}_1 x)$$

where $\mathbf{Z}_1$ and $\mathbf{Z}_2$ are new metavariables, and similarly for unifying $X(t_1, \ldots, t_n)$ and S.

If S is a term containing a variable z forbidden to X, to unify Xz and S, we take $\mathbf{Z} = \lambda z.(S \vee \mathbf{Z}z)$. The variable z will be forbidden to $\mathbf{Z}$ in the output environment, along with any other variables forbidden to X.

[1] That is, $t\sigma = s\sigma$ for some values of the the variables not in E.

The other clauses in the definition of unification are of two kinds: First, there are clauses similar to those for Robinson's original unification, but it should be noted that in the main recursive clause, where unification is applied successively to the arguments of a term, the output environment from each recursive call is the input environment when unifying the next argument. Second, there are clauses designed to ensure that terms which can be reduced (either by β-reduction or **d**-reduction) are reduced before the above rules are applied. For the details, see [2]. The rules given here should be sufficient to understand the applications to circumscription.

For example, if we want to unify $\mathbf{Z}(c)$ with **false**, we will get

$$\mathbf{Z} = \lambda x.(x = c \; ? \; \mathbf{false} \; : \; \mathbf{Z}(x))$$

which intuitively says that $\mathbf{Z}(c)$ should be **false**, but on all values of x different from c, $X(x)$ is undetermined. The use of a new metavariable expresses "undetermined".

Remark. In [2], it is proved that with respect to this notion of unification, unique most-general-unifiers exist, just as they do for Robinson unification in first-order logic. We want to take this opportunity to explain how this result reconciles with the well-known fact that there is no unique unifier for Huet's notion of λ-unification. Here is the statement of the most-general unifier theorem from [2]:

Theorem 1 (Most general unifier). *Let E be an environment. Suppose that p and q are normal terms in λD. Suppose that for some substitution θ legal for E, $p\theta$ and $q\theta$ are identical. Then p and q unify, and the answer substitution is legal for E, and more general than θ.*

The point is, that the conclusion would not be valid if we replaced the hypothesis "$p\theta$ and $q\theta$ are identical" by the hypothesis "$p\theta$ and $q\theta$ have a common reduct." Huet's terms with many unifiers do not form a counterexample to the theorem.

4 Blocks World Example

We treat the first example from [6] as a typical circumscription problem.

Let $\Gamma(Ab, On)$ be the theory

$$c \neq b \wedge \neg On(c) \wedge \forall x(\neg ab(x) \rightarrow On(x))$$

where the variables range over "blocks" and $On(x)$ means "x is on the table". Circumscription enables us to conclude that a is the only block not on the table. For simplicity, we first consider the problem without the predicate B, i.e. we assume all variables range only over blocks. The idea is that normal blocks are on the table, and since c is the only abnormal block, b is a normal block and hence is on the table. Circumscription should enable us to prove $On(b)$.

Circumscription in this example is taken to minimize ab with variable On, so in the general schema above, we take P to be ab and S to be On.

$$c \neq b \tag{1}$$
$$\forall x(\neg ab(x) \rightarrow On(x)) \tag{2}$$
$$\neg On(c) \tag{3}$$
$$\forall \Phi \Psi [\forall x(\neg \Psi(x) \rightarrow \Phi(x)) \wedge \neg \Phi(c) \wedge \Psi \leq ab \rightarrow ab \leq \Psi] \tag{4}$$

We first present a human-produced proof, for later comparison to the proof found by our program. We take as the goal to prove $On(b)$. Backchaining from (2) produces the new goal $\neg ab(b)$. The human then suggests the values

$$\Psi = \lambda x.(x = c \ ? \ \mathbf{true} : \mathbf{false}) \tag{5}$$
$$\Phi = \lambda x.(x = c \ ? \ \mathbf{false} : \mathbf{true}) \tag{6}$$

With these values of Φ and Ψ, we want to prove $ab(b) \rightarrow$ **false**, so we need to verify $\psi(b) =$ **false**. But $\psi(b) = (b = c \ ? \ \mathbf{true} : \mathbf{false})$, and $b = c$ evaluates to **false** since $b \neq c$ is in the antecedent, so $\psi(b)$ evaluates to **false**. It therefore suffices to verify the hypothesis of (4), namely

$$\forall x(\neg \Psi(x) \rightarrow \Phi(x)) \wedge \neg \Phi(c) \wedge \Psi \leq ab.$$

Fix an x, and suppose $\neg \Psi(x)$. Then $x \neq c$, from which $\Phi(x)$ follows, which proves the first conjunct. The second conjunct, $\neg \Phi(c)$, follows immediately by reduction to **true**. The third conjunct, $\Psi \leq ab$, is proved as follows: suppose $\Psi(x)$. Then $x = c$ and so we must prove $ab(c)$. But by (3) we have $\neg On(c)$, and so by (2) we have $ab(c)$. That completes the proof.

We now explain how the prover attacks this problem. We want to prove $\neg ab(b)$. (Officially that goal is the succedent of a sequent whose antecedent is the list of axioms.) So the prover assumes $ab(b)$, and the new goal is $ab(b) \Rightarrow$**false**. (Of course officially the axioms should appear in the antecedent of the goal sequent, too, but we do not write them.) This causes (3) to be "opened up", introducing metavariables $\mathbf{P}$ and $\mathbf{Q}$. The formula $ab \leq \mathbf{Q}$ is really $\forall w(ab(w) \rightarrow \mathbf{Q}(w))$, so a metavariable W is introduced for w as well, but soon it is instantiated to b to unify $ab(b)$ with $ab(W)$, in the hopes of proving $ab(W) \Rightarrow \mathbf{Q}(W)$ from $ab(b) \Rightarrow$**false**. Thus the prover tries to unify $\mathbf{Q}(b)$ with **false**. This gives

$$\mathbf{Q} = \lambda y.(y = b \ ? \ \mathbf{false} : Y(y))$$

where Y is a new variable. The next goal is the conjunction of the three formulae on the left of the implication in 4. These are taken in order; the first one is $\forall v(\neg \mathbf{Q}(v) \rightarrow \mathbf{P}(v))$. Fixing v the goal is $\neg \mathbf{Q}(v) \rightarrow \mathbf{P}(v)$; writing out the current value of $\mathbf{Q}$ and β-reducing, the goal is

$$\neg(v = b \ ? \ \mathbf{false} : Y(v)) \rightarrow \mathbf{P}(v)$$

There is a simplification rule for pushing a negation into a cases term, namely

$$\neg(v = p \ ? \ q \ : r) = (v = p \ ? \ \neg q \ : \neg r).$$

So the goal becomes

$$(v = b \ ? \ \mathbf{true} : \neg Y(v)) \rightarrow \mathbf{P}(v).$$

This is solved by second-order unification, taking

$$\mathbf{P} = \lambda v.((v = b ? \ \mathbf{true} : \neg Y(v)) \vee Zv).$$

The next goal is $\neg\mathbf{P}(c)$. That is, after a beta reduction,

$$\neg((c = b \ ? \ \mathbf{true} : \neg Y(c)) \vee Z(c))).$$

Now we can apply a simplification rule using the axiom $c \neq b$, reducing the cases term to $\neg Y(z)$ and hence the whole goal to $\neg(\neg Y(c) \vee Z(c))$. Using rewrite rules appropriate to classical logic we simplify this to $Y(z) \wedge \neg Z(c)$. Splitting the conjunction into two subgoals, the first one to be proved is $Y(c)$. This is solved by second-order unification, taking

$$Y = \lambda u.(u = c ? \ ab(b) : A(u))$$

where A is a new metavariable. You might think we should get **true** in place of $ab(b)$ in the value of Y, but when the prover has to prove a goal of the form $Y(c)$, it does not try to unify $Y(c)$ with **true**, but rather with one of the assumptions (formulas in the antecedent). It tries the most recently-added ones first, and it finds $ab(b)$ there, which explains the value given for Y.

The second goal is $\neg Z(c)$. Then $Z(c)$ is assumed, leading to a goal $Z(c) \Rightarrow$ **false**. Unifying $Z(c)$ with **false** gives Z the value

$$Z = \lambda r.(r = c \ ? \ On(c) : B(r)),$$

where B is a new metavariable. Again, you might expect **false** to occur in place of $On(c)$ in the value of Z, but the prover finds the value given, which is equivalent since $\neg On(c)$ is an axiom.

At this point, the values of $\mathbf{P}$ has become

$$\mathbf{P} = \lambda v.(v = b \ ? \ \mathbf{true} : \neg(v = c ? \ \mathbf{true} : A(z)$$

which simplifies to

$$\mathbf{P} = \lambda v.(v = b \ ? \ \mathbf{true} : v = c \ ? \ On(c) : \neg A(z))$$

The value of $\mathbf{Q}$ is now given by

$$\mathbf{Q} = \lambda y.(y = b \ ? \ \mathbf{false} : (\lambda u.(u = c \ ? \ ab(b) : A(u)))y)$$

which reduces to

$$\mathbf{Q} = \lambda y.(y = b \ ? \ \mathbf{false} : y = c \ ? \ ab(b) : A(y))$$

The next goal is $\mathbf{Q} \leq ab$, that is $\forall z(\mathbf{Q}(z) \rightarrow ab(z))$. Fixing z, the goal is $\mathbf{Q}(z) \rightarrow ab(z)$. Using the Gentzen rule for introducing $\rightarrow$ on the right, and writing out the current value of $\mathbf{Q}$, our goal is the sequent

$$z = b \text{ ? } \mathbf{false} : z = c \text{ ? } a(b) : W(x) \Rightarrow ab(x).$$

This is proved by cases, specifically by the cases-left rule.

Case 1, $z = b$. The goal reduces to **false** $\rightarrow ab(z)$ which is immediate. Case 2, $z \neq b$ and $z = c$. Then $\mathbf{Q}(a)$ reduces to

$$z = c \wedge z \neq b \wedge ab(b)$$

so the goal becomes

$$z = c, \quad z \neq b, \quad ab(b) \Rightarrow ab(z).$$

The human can note that $ab(c)$ follows from $\forall x(\neg ab(x) \rightarrow On(x))$ and $\neg On(c)$, and from $ab(c)$ the goal follows quickly. This is a relatively simple problem in first-order logic with equality, the difficulties of which are irrelevant to circumscription and second-order logic. Weierstrass is able to prove the goal.

Case 3, $z \neq b$ and $z \neq c$. Then $\mathbf{Q}(z)$ reduces to $W(z)$, so the goal becomes

$$W(z) \Rightarrow ab(z).$$

This goal is proved by instantiating the metavariable W:

$$W = \lambda z.(ab(z) \vee T(z))$$

where T is a new metavariable. The final values of $\mathbf{P}$ and $\mathbf{Q}$ are thus

$$\mathbf{P} = \lambda v.(v = b \text{ ? } \mathbf{true} : v = c \text{ ? } On(c) : (\neg ab(v) \wedge \neg T(v)))$$

$$\mathbf{Q} = \lambda y.(y = b \text{ ? } \mathbf{false} : y = c \text{ ? } a(b) : (ab(y) \vee T(y)))$$

To achieve the stated goal $On(b)$, the prover has only needed to deduce that b is not abnormal. Unlike the human, it has not gone ahead to deduce anything about other objects than a and b– the uninstantiated metavariable T remains as "undetermined". Of course, the constant b might as well have been a variable; the prover can prove $\forall x(x \neq a \rightarrow On(x))$ just as well as it can prove $On(b)$. But that proof, like the one above, will still use instantiations of $\mathbf{P}$ and $\mathbf{Q}$ involving free metavariables.

5 The automatically-produced proof

Here we present the proof as produced (and typeset) by our prover. In this proof, $a(x)$ stands for $ab(x)$, meaning "x is abnormal". $On(x)$, meaning "x is on the table", is represented by $o(x)$.

The goal is $o(b)$

Trying $\neg(a(X))$

Assuming $a(X)$
Trying for a contradiction
Trying left-arrow
Assume

$$\forall w(a(w) \rightarrow \mathrm{ap}(Q, w))$$

Still trying **false**
trying second-order unification (clause 1) on

$$\mathrm{ap}(Q, W) = \textbf{false}$$

The metavariable gets the value

$$Q = \lambda y. \begin{cases} \textbf{false} & \text{if} \quad y = W \\ \mathrm{ap}(Y, y) & \text{ow} \end{cases}$$

Trying $a(W)$
Aha! we have $a(W)$
Success
Success
Discharging

$$\forall w(a(w) \rightarrow \mathrm{ap}(Q, w))$$

Trying

$$\forall v(\neg(\mathrm{ap}(Q, v)) \rightarrow \mathrm{ap}(P, v)), \neg(\mathrm{ap}(P, c)), \forall z(\mathrm{ap}(Q, z) \rightarrow a(z))$$

Trying

$$\forall v(\neg(\mathrm{ap}(Q, v)) \rightarrow \mathrm{ap}(P, v))$$

Trying

$$\neg(\mathrm{ap}(Q, v)) \rightarrow \mathrm{ap}(P, v)$$

That reduces to:

$$\begin{cases} \textbf{true} & \text{if} \quad v = b \\ \neg(\mathrm{ap}(Y, v)) & \text{ow} \end{cases} \rightarrow \mathrm{ap}(P, v)$$

Assuming

$$\begin{cases} \textbf{true} & \text{if} \quad v = b \\ \neg(\mathrm{ap}(Y, v)) & \text{ow} \end{cases}$$

Trying $\mathrm{ap}(P, v)$
trying second-order unification (clause 2) on

$$\mathrm{ap}(P,v) = \begin{cases} \mathbf{true} & \text{if} \quad v = b \\ \neg(\mathrm{ap}(Y,v)) & \text{ow} \end{cases}$$

The metavariable gets the value

$$P = \lambda v.\vee\left(\begin{cases} \mathbf{true} & \text{if} \quad v = b \\ \neg(\mathrm{ap}(Y,v)) & \text{ow} \end{cases}, \mathrm{ap}(Z,v)\right)$$

Aha! we have $\mathrm{ap}(P, v)$
Success
Discharging
Success
Success
Trying

$$\neg(\vee(\neg(\mathrm{ap}(Y,c)), \mathrm{ap}(Z,c)))$$

Classically, it would suffice to prove:

$$\mathrm{ap}(Y,c), \neg(\mathrm{ap}(Z,c))$$

Trying $\mathrm{ap}(Y, c)$
trying second-order unification (clause 1) on

$$\mathrm{ap}(Y,c) = a(X)$$

The metavariable gets the value

$$Y = \lambda u.\begin{cases} a(b) & \text{if} \quad u = c \\ \mathrm{ap}(A,u) & \text{ow} \end{cases}$$

Aha! we have $\mathrm{ap}(Y, c)$
Success
Trying $\neg(\mathrm{ap}(Z, c))$
trying second-order unification (clause 1) on $\mathrm{ap}(Z, c) = o(c)$
The metavariable gets the value

$$Z = \lambda r.\begin{cases} o(c) & \text{if} \quad r = c \\ \mathrm{ap}(B,r) & \text{ow} \end{cases}$$

Aha! we have $\neg(\mathrm{ap}(Z, c))$
Success
That was the last conjunct, so the conjunction is proved.
Success
Trying

$$\forall z\left(\mathrm{ap}(\lambda y.\begin{cases} \textbf{false} & \text{if} \quad y = b \\ \mathrm{ap}(\lambda u.\begin{cases} a(b) & \text{if} \quad u = c \\ \mathrm{ap}(A, u) & \text{ow} \end{cases}, y) & \text{ow} \end{cases}, z) \to a(z)\right)$$

Trying

$$\mathrm{ap}(\lambda y.\begin{cases} \textbf{false} & \text{if} \quad y = b \\ \mathrm{ap}(\lambda u.\begin{cases} a(b) & \text{if} \quad u = c \\ \mathrm{ap}(A, u) & \text{ow} \end{cases}, y) & \text{ow} \end{cases}, z) \to a(z)$$

That reduces to:

$$\begin{cases} \textbf{false} & \text{if} \quad z = b \\ a(b) & \text{if} \quad z = c \to a(z) \\ \mathrm{ap}(A, z) & \text{ow} \end{cases}$$

Assuming

$$\begin{cases} \textbf{false} & \text{if} \quad z = b \\ a(b) & \text{if} \quad z = c \\ \mathrm{ap}(A, z) & \text{ow} \end{cases}$$

Trying $a(z)$
Trying left-arrow
Assume

$$\forall w(a(w) \to \mathrm{ap}(D, w))$$

Still trying $a(z)$
Failure
Discharging
Discharging
Proceed by cases:
Case 1:$z = b$
Then we have **false**
Trying $a(z)$
That case cannot occur
Case succeeded

```
              Case 2:$z = c$
              Then we have $a(b)$
                Trying $a(z)$
                  Trying left-arrow
                  Assume

                      $\forall w(a(w) \to \mathrm{ap}(D, w))$

                  Still trying $a(z)$
                  Failure
                  Discharging
                  Discharging
                Trying proof by contradiction of $a(z)$
                Assume $\neg(a(z))$
                  By axiom c2, it would suffice to prove $\neg(a(C))$
                    Trying $\neg(a(C))$
                    Aha! we have $\neg(a(C))$
                    Success
                Case succeeded
              Case 3:otherwise
                Trying $a(z)$
                trying second-order unification (clause 2) on

                        $\mathrm{ap}(A, z) = a(z)$

                The metavariable gets the value

                     $A = \lambda z.\vee(a(z), \mathrm{ap}(D, z))$

                Aha! we have $a(z)$
                Case succeeded
              Proof by cases succeeded
            Success
            Discharging
          Success
        Success
      That was the last conjunct, so the conjunction is proved.
      Success
    Success
  Success
Success. That completes the proof.
```

References

1. Beeson, M., *Foundations of Constructive Mathematics*, Springer-Verlag, Berlin/ Heidelberg/ New York (1985).
2. Beeson, M., Unification in Lambda Calculus with if-then-else, in: Kirchner, C., and Kirchner, H. (eds.), *Automated Deduction-CADE-15. 15th International Conference on Automated Deduction, Lindau, Germany, July 1998 Proceedings*, pp. 96-111, Lecture Notes in Artificial Intelligence **1421**, Springer-Verlag (1998).
3. Beeson, M., Automatic generation of epsilon-delta proofs of continuity, in: Calmet, Jacques, and Plaza, Jan (eds.) *Artificial Intelligence and Symbolic Computation: International Conference AISC-98, Plattsburgh, New York, USA, September 1998 Proceedings*, pp. 67-83. Springer-Verlag (1998).
4. Beeson, M., Automatic generation of a proof of the irrationality of e, in Armando, A., and Jebelean, T. (eds.): *Proceedings of the Calculumus Workshop, 1999, Electronic Notes in Theoretical Computer Science* **23** 3, 2000. Elsevier. Available at http://www.elsevier.nl/locate/entcs. This paper has also been accepted for publication in a special issue of *Journal of Symbolic Computation* which should appear in the very near future.
5. Beeson, M., Some applications of Gentzen's proof theory to automated deduction, in P. Schroeder-Heister (ed.), *Extensions of Logic Programming*, Lecture Notes in Computer Science **475** 101-156, Springer-Verlag (1991).
6. Doherty, P., Lukaszewicz, W., And Szalas, A., Computing circumscription revisited: a reduction algorithm, *J. Automated Reasoning* **18**, 297-334 (1997).
7. Ginsberg, M. L., A circumscriptive theorem prover, *Artificial Intelligence* **39** pp. 209-230, 1989.
8. *Introduction to Metamathetics*, van Nostrand, Princeton, N.J. (1950).
9. Lifschitz, V., Computing circumscription, in: *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, volume 1, pages 121-127, 1985.
10. McCarthy, J., Circumscription, a form of non-monotonic reasoning, *Artificial Intelligence*, **13** (1-2), pp. 27-39, 1980.
11. Przymusinski, T., An algorithm to compute circumscription, *Artificial Intelligence*, **38**, pp. 49-73, 1991.