Solving For Functions

Michael Beeson
San Jose State University
Math & Computer Science
San Jose, CA 95192

Abstract

Buchberger has emphasized that automated deduction involves computation, proving, and solving. When the object to be "solved for" is a function, second-order unification can be a very powerful and general solution tool. An algorithm for second-order unification was given in (8). This algorithm is now being implemented in the source code of Otter; this algorithm differs from the earlier algorithm given by Pietrzykolski (22). Several examples are given to illustrate the wide range of potential applications of this algorithm and its implementation in a powerful clausal theorem prover, including the ability to manipulate quantifiers at the clausal level, so that definitions involving quantifiers can be conveniently used in proofs. Since predicates are considered as Boolean-valued functions, solving for functions includes solving for predicates, and second-order unification can help first-order provers with proofs by induction. Other examples show proofs in which the function to be solved for is a one-to-one correspondence (in set theory) or a group isomorphism. There is also a detailed comparison of the new second-order unification algorithm with the older one.[*]

KEYWORDS: automated deduction, computer proofs, unification, second-order, Otter

Introduction

In previous work (4; 7), the author has focused attention on the relation between computation and logic in automated deduction. In the development of *Theorema*, Buchberger (13) was guided by his vision that mathematics involves *three* essential components: logic (proving), computation, and *solving*. Solving

[*]Research supported by NSF grant number CCR-0204362.

means finding an x with certain desired properties. Often the "solving" step of a proof is the "key" step, the one that seems to demand "creativity". Buchberger emphasizes the importance of adding (many) special-purpose "solvers" to a computerized mathematical system. In this paper, by contrast, we wish to focus on *general* solution techniques, as opposed to special algorithms that apply to a specific mathematical theory.

Solving refers to techniques for instantiating a variable. The traditional method of instantiating variables used in automated deduction is unification. It is known that, with respect to first-order logic, this method is "universal", in the sense that there are completeness theorems for various systems of inference in which unification is used to instantiate the variables. For example, binary resolution with unification is complete, and backwards application of the Gentzen sequent-calculus rules, guided by unification, is complete. But when we go to mathematics, unification does not seem to do the job. For example, if we want an x such that $x^2 + ax + b = 0$, we need the quadratic theorem, not unification. Therefore, as Buchberger has emphasized, special-purpose solvers are required to deal with the different specialized branches of mathematics.

Our focus in this paper is on situations where we need to "solve for" a function, rather than an element or object. We will exhibit a number of examples of proofs of this kind, to illustrate the claim that "solving for a function" is a theme that permeates different branches of mathematics. Just as this is a general theme in mathematics, there is a general tool in logic to help with this kind of proof. There is a well-known "unification algorithm" which can be thought of as solving equations between terms denoting objects. There is also λ-calculus, which lets us define terms for denoting functions. There is a way (in fact, more than one way) to generalize the unification algorithm so that it can be thought of as solving equations between terms denoting functions, rather than objects. This is known as *second-order unification*.

Our motivation in this paper is to show that, even though special-purpose solvers are useful for specialized mathematics, we can get some surprising (and in some cases unexpected) results from general methods based on second-order unification.

The problems addressed here are: How are the existing notions of second-order unification related? Can second-order unification be fruitfully used in a first-order theorem-prover? Which is the best notion of unification to use in automated deduction? Is it feasible to add second-order unification to an existing prover (Otter), which already has a large group of users? How can we expect this capability to be useful in automated deduction?

These questions have not been answered until now. One reason for this is that the majority of work in automated deduction has been done (so far) by first-order theorem provers, but second-order unification has so far been (incorrectly) viewed as incompatible with first-order provers. Another reason is that serious automated deduction has so far been done in theories with a single short list of axioms, referring to only one kind of mathematical objects, rather than in more complex mathematical environments, where second-order unification might prove helpful. A third reason is that second-order unification is considered inefficient (it produces infinitely many unifiers, it necessarily produces redundant unifiers, it involves an exponential search, etc.)

The main points of the paper are

(i) Second-order unification can be thought of as "solving for a function", that is, finding a term that defines a function with desired properties to complete a proof.

(ii) Some mathematical problems that may not appear to have the form of "solving for a function" can be recast in that form, so that second-order unification can be used on them. Others are naturally of that form. A very general "solver" can be useful, because solving for functions occurs generally in mathematics.

(iii) Second-order unification can be implemented and used in Otter, a first-order theorem-prover already widely used (19). There is nothing strange or difficult about using λ-calculus and "second-order" unification in a first-order prover. Sample proofs produced by the new implementation in Otter are exhibited.

(iv) Technical difficulties concerning efficiency, non-termination, redundancy, etc., are minimized if we use the notion of unification introduced in (8) rather than the one in (22; 18). These notions are compared here.

(v) Quantification can be defined in terms of λ-calculus, and second-order unification makes it possible to use quantifiers in Otter proofs at the clausal level. This is very important as it will enable Otter to work with definitions involving quantifiers, such as the relation "u divides v" on the natural numbers.

(vi) Set theory too is naturally treated as a branch of λ-calculus. This is not an original idea but goes back to Church.

We shall give a motivating example. In undergraduate courses in algebra, what is taught as "group theory" usually involves the study of groups and subgroups. See e.g. the first 45 pages of (15) for the mathematics in question. The notation a^n is introduced, where n is a natural number, and one of the early theorems is Lagrange's theorem, according to which the order of a subgroup H of a finite group G divides the order of G. This theorem involves groups, subgroups, and natural numbers. Its proof begins by showing that each coset Ha is in one-to-one correspondence with H. That is, there exists a function f mapping H one-to-one onto Ha. That function, of course, is $\lambda x.xa$. The proof therefore involves a small amount of set theory to deal with cosets and one-to-one correspondences, as well as enough number theory to deal with "divides". We will show in the last section of this paper that second-order unification can instantiate the variable f properly to do the key step of this proof automatically.

Buchberger's aim in developing *Theorema* has been to develop an interactive environment in which humans can develop computer-checked proofs in a mathematical context like this, in which (elements of) several different branches of mathematics are available at the same time. One difficulty in such an enterprise is that the proof-checker may require an unacceptably large level of detail. People refer to the "expansion factor", by which a page of proof written by and for humans expands to ten or more pages of computer-checkable proofs. If the details could be taken care of automatically, that would advance the subject. On the other hand, some researchers in automated deduction have focused on the attempt to have the computer prove results not previously proved by humans. So far, these efforts have been successful only in areas that can be axiomatized by a few simple axioms, and studied in isolation from the rest of mathematics. We view our work as directed towards opening up wider horizons to automated deduction in the future, not necessarily just as support for proof-checking.

Solving for functions in mathematics

In this section, we will illustrate the theme that "solving for functions" occurs across the board in different branches of mathematics. We have already given the example of Lagrange's theorem in algebra. The context of Lagrange's theorem is typical of a great deal of mathematics: a little set theory, a little number theory, sometimes a little calculus. Two kinds of objects are considered (numbers and elements of the group), along with sets of objects (subgroups), functions from objects to objects (isomorphisms, etc.), and functions from objects to numbers (the order of an element), and even functions from sets of objects to numbers (the order of a group, the index of a subgroup). Objects of greater complexity than that are not required. The recently announced polynomial-time primality test (1), for example, which was hailed in the *New York Times* (Aug. 8, 2002) as the best result in computer science in the past ten years, is actually mathematics of the sort just described. One needs integers, integers mod p, and polynomials over Z_p, groups and their orders, but nothing more complicated. Fermat's conjecture and its long and complex proof notwithstanding, if automated deduction could deal successfully with the simple kinds of contexts just described, the subject would advance rapidly.

We begin with set theory. Consider $S = \{n : P(n)\}$, where P is some property of integers. For simplicity let us just consider sets of integers. The characteristic function χ_S is defined by $\chi_S(n) = 1$ if $n \in S$, 0 if $n \notin S$. What is the relation between χ_S and S? The logician Alonzo Church suggested that in fact the set *is* just its characteristic function. (See the Appendix of (2).) According to Church, any set is a boolean-valued function, and $n \in S$ is just an abbreviation for $\chi_S(n) = \mathbf{true}$. If we take this view, sets are functions, and can be defined by λ-terms. Finding a set with certain desired properties becomes a special case of solving for functions. The "list notation" $\{a, b, c\}$ for a finite set can be regarded as an abbreviation for a function defined by cases: if x is a,b, or c, the value is **true**, else it is **false**.

One can also use tuple notation $\langle x, y\rangle$ and define Cartesian products. Since our interest is not foundational, we ignore the issue of whether and how tuples can be defined in terms of sets. We assume that $\mathbf{p_0}$ and $\mathbf{p_1}$ are pairing functions such that $\langle \mathbf{p_0} z, \mathbf{p_1} z\rangle = z$. We say $A \cong B$ if there is a one-to-one correspondence between A and B. A simple theorem in this subject (just about the simplest set-theoretical theorem I can think of) is that $A \times B \cong B \times A$. ("$\cong$" is read "equipollent"). How do we prove that theorem? By solving for a function f that maps $A \times B$

one-to-one onto $B \times A$. We might like to write that function as $\lambda\langle x, y\rangle.\langle y, x\rangle$, but according to the usual notation for λ-terms, that is not a syntactically correct term. What we want would be formally written as $\lambda z.\langle \mathbf{p_1}(z), \mathbf{p_0}(z)\rangle$. This term arises naturally as the solution to a certain second-order unification problem, but there is no space here for the details.

Turning to number theory, let us consider the relation $n|m$. To define this set (or its characteristic function), we need an existential quantifier: $n|m$ means $\exists k(n * k = m)$. We shall take up this example in a subsequent section, and show that when the existential quantifier is defined using λ-calculus, second-order unification can use this definition to verify that $2|6$ automatically.

Consider a related but more difficult example, the theorem that two positive integers n and m have a greatest common divisor. That is, there exists a number k such that k divides both n and m, and any other integer j that divides both n and m also divides k. We know, from our mathematical education, two different ways to instantiate k: as the least number of the form $\lambda n + \mu m$, where λ and μ are (positive or negative) integers; or as the result of the Euclidean algorithm $E(n, m)$. It seems that neither definition of k will be produced by unification, even though the existence of gcd's follows in first-order logic from the first-order version of Peano's axioms. Is that a slight mystery, since unification is complete for first-order logic? It should not be: Let us abbreviate by $Q(n, m, e)$ the formula that says e is the gcd of n and m. Then there are certain instances of induction (say for simplicity there is just one)

$$I := \forall n(P(n) \to P(n+1)) \to \forall m P(m)$$

such that in first order logic, $I \wedge Z- > \forall m, n \exists e Q(n, m, e)$. Here Z is the conjunction of the non-induction axioms of arithmetic. Once the proper formula or formulas P is discovered, then resolution driven by unification can, of course, find the proof. Our point here is that finding this proof involves as its main task the discovery of the appropriate instance of induction to use. Now the property P is just a boolean-valued function, and therefore second-order unification can (potentially) find it. We would try to unify the conclusion of the instance of induction, $P(m)$, with the theorem to be proved, in order to find an instantiation of P. Here P would be regarded as a variable (for a function).

We seem to need two sorts or types: numbers and functions. But several sorts (or even infinitely many) can be reduced to first-order, using in this case a unary predicate $N(x)$ for the numbers. We could, if we liked, use another predicate for functions from numbers to booleans, etc.[†]

[†]The distinction between first-order and higher-order logics is not as important as many people assume. Syntactically the two are intertranslatable, using unary predicates to distinguish the types. The difference arises in the *semantics*. The second-order Peano axioms, for example, have an instance of induction for each subset of the integers, of which there are uncountably many. The first-order version of the Peano axioms has induction only for those subsets that can be defined by first-order formulas in a fixed language. Many instances of induction are thus omitted–hence there are non-standard models. There are no non-standard models of the second-order Peano axioms, as Peano himself proved: these axioms characterize

People say that Otter "can't do induction". But induction, with respect to first-order formulae, just involves deduction from first-order axioms, and Otter is as good at that as at any other kind of first-order logic. If the user is willing to specify what instances of induction are required, those instances can be given as axioms to Otter, suitably Skolemized, and Otter can find the proof, at least in some examples I have tried. What Otter cannot do is find the correct instance of induction. This is a problem in "solving for functions", since the instance of induction to be found is a boolean-valued function. In some cases, second-order unification can find the required instance.

One problem here is that we often want to prove a quantified statement by induction. For example in the case of the existence of gcds, the conclusion $Q(n, m, e)$ involves a quantifier. The general plan in automated deduction has been to replace quantifiers by functions. Usually this is done by using symbols for Skolem functions. But if λ-terms are in use, it is possible to deal with quantifiers more directly, using second-order unification to guide the proof. We amplify this point in detail in a subsequent section.

Second-order unification

We focus on the attempt to extend unification to instantiate variables for functions by means of λ-terms. This is loosely known as *higher-order unification.* It has been pursued in the past in the context of formal systems based on typed λ-calculus, so that functions and functionals of any "finite type" can be considered. We call our unification "second-order" since, formally, we prefer a framework without types. This is not essential; our work can be embedded in several different formalisms. In (22), Pietrzykolski gave an algorithm for second-order unification, which we here call λ-unification. This was extended to type theory in (23). Huet showed in (18) that λ-unifiability is more efficient than λ-unification, and subsequently λ-unifiability was used in the implementation of Coq. Huet proved a completeness theorem for λ-unification relative to typed λ-calculus.

λ-unification, and the completeness theorem for it, are for typed λ-calculus without definition by cases. If we try to use λ-unification to find an f such that $f(0) = 0$, it will give us two answers: $f = \lambda x.0$ and $f = \lambda x.x$. If we then ask for an f such that $f(0) = 0$ and $f(1) = 2$, neither of these solutions will do, and indeed no solution is given by a pure λ-term. However, if we look for solutions involving if-then-else, it is easy to define such a function. This example is a fairly representative one, illustrating the two methods "projection" and "imitation" used in the definition of λ-unification.

In (8), a notion of second-order unification is given that allows the construction of functions using definition by cases. Here we call this notion D-unification, to distinguish it from λ-unification. We use $\mathbf{cases}(n, m, x, y)$ to mean "if $n = m$ then x, else y". This unification involves making the "minimal commitment" needed to meet the conditions. Thus, if we use it to solve the problem of finding f such

the integers up to isomorphism. But semantics are not important for automated deduction, which is an inherently syntactic enterprise.

that $f(0) = 0$, the answer we get is $f = \lambda x.\mathbf{cases}(x, 0, 0, Y(x))$, where Y is a fresh variable. Intuitively this means, "if $x = 0$ then 0, else undetermined". Here "undetermined" is different from "undefined", but the idea is similar to "not yet defined". It means that further values of f can be specified by instantiating the new variable Y. This answer is "more general" than the two answers $\lambda x.0$ and $\lambda x.x$ given by Huet's algorithm, in the sense that they can be obtained from it by instantiating Y in different ways. In (8), a most-general-unifier theorem is proved, according to which, if terms t and s (say in typed λ-calculus with definition by cases) unify at all, then they have a unique most general unifier. This is a satisfying generalization of the most-general-unifier property of Robinson's "ordinary" unification.

An important idea in D-unification is that of *restrictions* on a variable. A restriction is a pair consisting of a variable and a (possibly empty) list of constants. Note that if a problem expressed in first-order logic using quantifiers is converted to clausal form, some of the originally-bound variables are converted to constants, so the concept "constant" here includes what sometimes are called "object variables", and the concept "variable" here includes what is sometimes called "metavariable". The idea is that the list of constants paired with x are *forbidden to* x.[‡] Unification is not allowed to assign a variable x a value that contains a constant forbidden to x. To make this sensible, the input to the unification algorithm has to include an *environment*, which is a finite list of restrictions. Since D-unification can introduce new variables, not mentioned in the input environment, the output of the unification algorithm is not only a substitution but also an *output environment.* The substitution σ unifies t and s relative to environment E if for some substitution χ whose restriction to E is the identity, we have $t\sigma\chi = s\sigma\chi$. That is, it is possible to extend σ to give the new variables values such that, under the extended substitution, t and s become equal.

We will state the most-general-unifier theorem, which is the nicest property of D-unification. Before the theorem can be stated, the notion of "most general substitution" must be defined. Here are the definition and the theorem, taken from (8).

Definition: Given an environment E, θ is more general than μ, relative to E, if there is a substitution β such that $\theta\beta = \mu$ on E. That is, for all variables X in the environment E, we have $X\theta\beta \cong X\mu$. Here $p \cong q$ means $px = qx$, where x is a variable not in p or q.

Remark. In this definition, the phrase "$px = qx$" means that $px = qx$ is provable in the λ-calculus with definition by cases (be it a typed version or the untyped theory λ-D considered in (8)).

[‡]Technically, one has to decide whether x in $\lambda x.t$ is a variable or a constant. Conceptually it is constant, since unification cannot assign it a value. But if one takes that seriously, one must constantly be replacing variables by constants and vice-versa when β-reduction removes λ. In the Otter implementation we do not do this.

THEOREM 0.1: [Most general unifier] *Let E be an environment and p and q normal terms. Suppose that for some substitution θ legal for E, $p\theta$ and $q\theta$ are identical. Then D-unification terminates successfully on inputs E,p, and q, and the answer substitution is legal for E, and more general than θ.*

Comparison of D-unification and λ-unification

The uniqueness of the result of D-unification, and the most-general-unifier theorem, may seem puzzling in view of the many-valuedness of λ-unification. As the author of (8), I received a number of inquiries about this point. Is λ-unification many-valued only because of examples like the one above (in which it seems that the reason is over-specification of the unifier), or also for other, possibly more fundamental reasons? How do the two theorems (completeness of Huet's algorithm and most-general-unifier theorem) avoid contradicting each other? The purpose of this section is to answer these questions.

Let us consider the statement of the theorems carefully. The hypothesis of the most-general-unifier theorem of (8) is that t and s are given normal terms, and there is a substitution θ such that $t\theta$ and $s\theta$ are identical. In that case, says the theorem, there is a most general unifier. The terms t and s, as well as the unifier, can involve definition by cases. What if we replace the hypothesis that $t\theta$ and $s\theta$ are identical by the weaker requirement that $t\theta$ and $s\theta$ are provably equal (i.e., in view of Church-Rosser, they have the same normal form)? Is the theorem still true? Also, if t and s are provably equivalent in λd-calculus, are they necessarily unifiable? These questions are not answered in (8).

The completeness theorem for λ-unification has the weaker hypothesis, that $t\theta$ and $s\theta$ have the same normal form, but neither the terms t and s nor θ can involve **cases**. The conclusion is that there is a complete set of unifiers (a CSU) for t and s, i.e. every unifier is more general than some substitution in the CSU. The CSU can be infinite.

It will be instructive to consider an example from (22), which was also considered in (18), and compare how the example is treated by λ-unification and D-unification. The example is the unification of $F(F(X))$ with $a(a(b))$. Here F and X are variables and a and b are constants. **cases**-unification finds (only) the solution $\{F := a, X := b\}$. λ-unification finds this solution, and also the solution $\{F := \lambda u.a(a(b)), X := b\}$ (here F is a constant function) and the solution $\{F := \lambda u.u, X := a(b)\}$ (here F is the identity function). The details of the calculation for λ-unification are on p. 42 of (18).[§] The example brings to the fore the fact that D-unification has no clause in its definition that applies to the case of $F(t)$, where t is a compound term containing variables forbidden to F, such as (in this case) F itself. Therefore only the "Robinson clause" (which is the same as first-order unification) applies to this example, which is why we get only the first solution.

[§]There is a technicality about whether η-reduction is used in λ-unification or not. It is used in (22). If it is not used, as in (18), we get one more solution, $\{F := \lambda u.a(u), X := b\}$, which is η-equivalent to $\{F := a, X := b\}$.

This example shows that the most general unifier theorem for D-unification depends critically on the hypothesis that $t\sigma$ and $s\sigma$ are identical, not just β-convertible. To see this, take t to be $F(F(X))$ and s to be $a(a(b))$. Take σ to be the second solution substitution above. Then $t\sigma$ and $s\sigma$ are beta-convertible, but it is not the case that σ is more general than $\{F := a, X := b\}$; indeed since this substitution has constants on the right, it is not more general than any other substitution.

An even simpler example can be given to show that the most general unifier theorem fails if we change the hypothesis to "$t\sigma$ and $s\sigma$ are β-convertible" instead of "identical". Consider the unification problem $X(t) = t$, where t is a compound term. Then D-unification does not succeed, as there is no clause in the definition that applies. But if we take θ to be the substitution $\{X := \lambda x.x\}$, then $X(t)\theta$ is β-convertible to $t\theta$. This would be a counterexample to the theorem with the changed hypothesis.

Implementation in Otter

D-unification has earlier been implemented in a backwards-Genzten theorem prover (9). However, that prover is not as strong or robust as Otter, and the well-known powers of Otter should work well in combination with D-unification. Implementation of this algorithm in the source code of the theorem-prover Otter is well under way at the time of writing (August 2002). There are some additional factors: Otter already has a large user community, so implementation in Otter will make D-unification readily available, without anyone having to learn a new system. Also, Fitelson and Harris have written a Mathematica interface to Otter (private communication), which can be used to connect *Theorema* to Otter.

The implementation had to begin with adding λ-calculus to Otter. Reserved words `lambda` and `ap` (or synonymously `Ap`) are used for this purpose. One uses `lambda(x,ap(f,x))` to enter the term $\lambda x.f(x)$. Beta-reduction has been implemented in the framework Otter uses for demodulation. Technicalities necessary to avoid clash of bound variables have been successfully dealt with. The following Otter proof shows a beta-reduction combined with an ordinary demodulation, given the demodulator $x * x = x$. The theorem proved is

$$(\lambda x.x * x)c = c.$$

Of course the proof is trivial: it is only meant to demonstrate the successful implementation of β-reduction working more or less the same way as ordinary demodulation in a first-order theorem-prover. Line 3 of the proof is the negation of the goal. The first two lines are axioms. The proof completes with the derivation of a contradiction.

```
1 [] x=x.
2 [] x*x=x.
3 [] ap(ap(lambda(x,lambda(y,x)),c*c),y)!=c.
4 [3,demod,2,beta,beta] c!=c.
5 [binary,4.1,1.1] .
```

Quantification in a clausal theorem-prover

In this section we show how to treat quantification in Otter, at the clause level, based on the machinery of λ-calculus and second-order unification. An example proof will be worked through in detail.

One can regard $\exists$ as a boolean-valued functional, whose arguments are boolean functions (defined, for simplicity, on the integers, let's say). Then $\exists n.P(n)$ is an abbreviation for $\exists(\lambda n.P(n))$, or more explicitly, $Ap(\exists, \lambda n.P(n))$. It is then possible to work with quantified statements directly in Otter–that is, in the version of Otter that is enhanced with λ-calculus.

We will show exactly how this is done. In the presence of λ-calculus, $\exists$ is treated as a constant. The rule for dealing with $\exists$ in Otter is

```
-Ap(Z,w) | exists(lambda(x, Ap(Z,x))
```

This works in Otter as follows: if $Z(t)$ can be proved for any term t, then the literal $-Ap(Z, x)$ will be resolved away, using the substitution $x := t$. Then $\exists(\lambda x.Z(x))$ is deduced, which can be abbreviated to $\exists x.Z(x)$. No machinery is added to Otter to accomplish this, other than what has already been added for λ-calculus.

This will permit the use of definitions that explicitly involve a quantified formula in the definition. For example, we could define

```
divides(u,v) = exists(lambda(x,u*x = v)).
```

Now, given the clause $2 * 3 = 6$, Otter can deduce `divides(2,6)` as follows: First, the negated goal `-divides(2,6)` will rewrite to
`-exists(lambda(x,2*x = 6))`.

This will unify with `exists(lambda(x,Ap(Z,x))` if `2*x=6` will unify with unify with `Ap(Z,x)`. Second-order unification (with x forbidden to Z) will find

$$Z = \lambda w.(2 * w = 6 \vee Y(w))$$

where Y is a new variable. Resolution of the two clauses containing `exists` will then generate a new clause containing the single literal $-Ap(Z, w)$ with this value of Z. Specifically,

$$-Ap(\lambda w.(2 * w = 6 \vee Y(w)), w).$$

That will be β-reduced to $-(2 * w = 6 \vee Y(w))$ so the clause finally generated will be `-or(2*w = 6, Y(w))`. Demodulation can apply to a negated literal under these circumstances and produce two new clauses, `-(2*w = 6)` and $-Y(w)$.¶ The first of these resolves with $2 * 3 = 6$, producing a contradiction that completes the proof.

¶Demodulation technically leads from term to term, or clause to clause. We refer here to a generalized version of demodulation which treats certain functors, such as `or`, specially. It also incorporates β-reduction.

Two simple examples

The first example is the following theorem: If $a \neq b$, there exists a predicate which is true on a and false on b. The input file contains some equations for **cases** as well as the clauses `a != b` and `-Ap(X,a) | Ap(X,b)`, which is the way we write $\neg X(z) \vee X(b)$ in Otter. The inference rules are binary resolution and paramodulation. The binary resolution rule has been modified so that a term $X(a)$ or its negation can always be unified with the constant `$F`, which is Otter's name for falsity. Unifying `-Ap(X,a)` with `$F` Otter finds `X = lambda(x,cases(x,a,$T,Ap(Y,x))` , where `Y` is a new variable. The result of the binary resolution step is `Ap(lambda(x,cases(x,a,$T,Ap(Y,x)),b)`. This beta-reduces to `cases(b,a,$T,Ap(Y,b))`. But before this, Otter has already deduced `cases(b,a,x,y) = y`. Since we have set the option `knuth-bendix`, which turns on paramodulation, this previously-deduced equation is used as a demodulator, and our new clause demodulates to `Ap(Y,b)`. This clause then unifies with `$F`, producing `Y = lambda(y,cases(y,b,$F,Ap(Z,y))`. The resolvent is the empty clause, completing the proof. The value of the function found is, after beta-reduction, `X = lambda(x,cases(x,a,$T,cases(x,b,$F, Ap(Z,x))))`. Intuitively, this function is true on a, false on b, and elsewhere undetermined.

Our second example is to prove the existence of the identity function. The goal is $\exists F \forall x(x = Ap(F,x))$. Traditionally, to formulate a problem of the form $\exists x \forall y P(x,y)$ for Otter, one would introduce a Skolem function g, and write the negated goal as $\neg P(x, g(x))$. By comparison, in a backwards Gentzen prover (such as was used in (9)), one would change y to a constant forbidden to x, and the negated goal (which one does not have to negate in such a prover, but we negate it for comparison to Otter) would be $\neg P(x,c)$, with c forbidden to x. Intuitively, these two goals are similar, since $g(x)$ functions more or less the same as a constant forbidden to x. It is like an arbitrary constant because the Skolem function is not further specified–$g(x)$ could be any arbitrary value–and it is "forbidden to x" by the occurs check in unification.

We take the negated goal in the form $\neg P(x,c)$, with c forbidden to x. In our example this is $c \neq Ap(F,c)$. We resolve this with the equality axiom $x = x$. First x is given the value c. Then we have to unify $Ap(F,c)$ with c, where c is forbidden to F. The definition of D-unification tells us $F := \lambda x.(x \vee Yx)$, where Y is a new variable. The succesful unification derives a contradiction by binary resolution and completes the proof of the theorem.

On the other hand, if we take the Skolemized version of the negated goal, we have to unify $Ap(F, g(F))$ with $g(F)$, where g is a Skolem function. We have discussed this shortcoming (being unable to unify $X(t)$ with t) in an earlier section. Until this difficulty is overcome, we use constants forbidden to F when converting a theorem to clausal form.

A more mathematical example: Lagrange's theorem

In this section we use D-unification to work out the most important details of the proof of the algebraic part of Lagrange's theorem: the existence of a one-to-one correspondence between H and the coset Ha, where $a \in G$.

To prepare this for Otter, we use a unary predicate $G(x)$ for the group G, a unary predicate $H(x)$ for the subgroup H, and include the axioms that assert that G is a group under $*$ with identity e and inverse $i(x)$, and that H is closed under $*$ and inverse, and $H(x)$ implies $G(x)$. We also include the following:

```
 -Ap(Z,w) | exists(lambda(x,Ap(Z,x))).
G(a).
```

Now consider how to formalize the coset Ha. We have

$$\begin{aligned} Ha &= \{ha | h \in H\} \\ &= \{w : \exists h.(h \in H \wedge h * a = w\} \\ &= \lambda w.(\exists h.(h \in H \wedge h * a = w) \\ &= \lambda w.(\exists(\lambda(h, H(h) \wedge h * a = w))) \end{aligned}$$

We put this into Otter using a function symbol `ha`, as follows:

```
ha(w) = exists(lambda(h, and(H(h), h*a = w))).
```

We use this as a demodulator, so that it will be used to rewrite any literal -ha(t) that arises. Such a literal would then resolve with existential axiom, using D-unification, and the resulting substitution would yield

```
Z:= lambda(h, or(and(H(h),h*a = t), Ap(Y,h)))
```

where Y is a new variable.

Now, for simplicity, we begin by proving that there is a function F from H to Ha, without worrying about the one-to-one and onto part yet. The goal is then

$$\exists F \forall x(x \in H \rightarrow Ap(F, x) \in Ha)$$

Instead of Skolemizing $x = g(F)$, we replace x by a constant that is forbidden to F. The negated goal becomes the two clauses

```
H(c).
-ha(Ap(F,c)).
```

As shown above, the literal `-ha(Ap(F,c))` demodulates and resolves with the existential axiom; the unification produces the substitution

```
Z:= lambda(h, or(and(H(h),h*a = Ap(F,c)), Ap(Y,h)))
```

and the resolution produces the unit clause

```
-Ap(lambda(h,or(and(H(h),h*a = Ap(F,c)), Ap(Y,h))),w).
```

This clause however is not stored yet, because it β-reduces to

```
-or(and(H(w),w*a = Ap(F,c)),Ap(Y,w))).
```

Demodulators are used to implement de Morgan's laws, so this will demodulate to the two clauses `-H(w) | w*a != Ap(F,c)` and `-Ap(Y,w)`. Now we're getting somewhere! The literal `-H(w)` resolves with `H(c)`. The inferred clause is $c * a! = Ap(F, c)$. The single literal in this clause can be unified with the equality axiom $x = x$. This results in unifying $c * a$ with $Ap(F, c)$. Since c is forbidden to F, the answer substitution is $F := \lambda x.x * a \vee Y(x)$. If we take $Y = \lambda x.\mathbf{false}$ we have $F = \lambda x.x * a$ (since $u \vee \mathbf{false} = u$, even if u is not Boolean). $\lambda x.x * a$ is the desired map from H to the coset Ha. The proof that it is one-to-one and onto is relatively straightforward.

Related work

The pioneer of the actual use of second-order unification was Huet (18). It has been implemented in Coq, which is described in (3) as well as numerous documents available from the Coq web page. There are many existing implementations of higher-order logic, including: λ-prolog (20); PVS, which is being used at SRI under the direction of N. Shankar (21); HOL-Light, which was written by John Harrison (16; 17) and is currently being used by him at Intel's Portland facility; NuPrl, developed at Cornell under the direction of Constable (14); and the French system Coq developed at INRIA, which is also being used in Nijmegen. These systems (if they implement higher-order unification at all) use λ-unification, rather than D-unification, and they are primarily proof-checkers, not proof-finders. We believe that D-unification offers greater power and efficiency, as does the use of the industrial-strength clausal theorem prover Otter. A proof of Lagrange's theorem has been checked by the use of Nqthm (25).

Some years ago, Quaife used Otter to formalize set theory using a finite axiomatization due in essence to Gödel-Bernays. He then used set theory to formulate Peano arithmetic, and he succeeded in proving the fundamental theorem of arithmetic (24). Belinfante, building on Quaife's beginning, has carried forward the formal development of set theory from first principles, proving thousands of theorems in Otter (10; 11); see also Belinfante's web site.

References

[1] Agrawal, M., Saxena, N., and Kayal, N., PRIME is in P, available from `http://www.cse.iitk.ac.in/news/primality.html`

[2] Barendregt, H., *The Lambda Calculus: Its Syntax and Semantics*, Studies in Logic and the Foundations of Mathematics **103**, Elsevier Science Ltd. Revised edition (October 1984).

[3] Barendregt, H., and Geuvers, H., Proof-Assistants Using Dependent Type Systems, in: Robinson, A., and Voronkov, A. (eds.), *Handbook of Automated Reasoning, vol. II*, pp. 1151-1238. Elsevier Science (2001).

[4] 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).

[5] Beeson, M., *Mathpert*: Computer support for learning algebra, trigonometry, and calculus, in: A. Voronkov (ed.), Logic Programming and Automated Reasoning, Lecture Notes in Artificial Intelligence 624, Springer-Verlag (1992).

[6] Beeson, M., *Mathpert Calculus Assistant.* This software product was published in July, 1997 by Mathpert Systems, Santa Clara, CA. See www.mathxpert.com to download a trial copy.

[7] 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).

[8] 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).

[9] Beeson, M., A second-order theorem prover applied to circumscription, in: Gor, R., Leitsch, A., and Nipkow, T. (eds.), *Automated Reasoning, First International Joint Conference, IJCAR 2001, Siena, Italy, June 2001, Proceedings*, Lecture Notes in Artificial Intelligence **2083**, Springer-Verlag (2001).

[10] Belinfante, J., Computer proofs in Gödel's class theory with equational definitions for composite and cross, *J. Automated Reasoning* **22**, No. 3 (1988), pp. 311-339.

[11] Belinfante, J., On computer-assisted proofs in ordinal number theory, *J. Automated Reasoning* **22**, No. 3, pp. 341-378.

[12] Boyer, R. S., and Moore, J. S., *A Computational Logic Handbook*, Academic Press, Boston (1988).

[13] Buchberger, B., *et. al.* Theorema: An Integrated System for Computation and Deduction in Natural Style, in: *Proceedings of the Workshop on Integration of Deductive Systems at CADE-15, Lindau, Germany, July 1998*

[14] Constable, R. L. *et. al.*, *Implementing Mathematics with the Nuprl Proof Development System*, Prentice-Hall, Englewood Cliffs, New Jersey (1986).

[15] Hall, Marshall, Jr., *The Theory of Groups*, Macmillan, New York (1959).

[16] Harrison, J., and Théry, L.: Extending the HOL theorem prover with a computer algebra system to reason about the reals, in *Higher Order Logic Theorem Proving and its Applications: 6th International Workshop, HUG '93*, pp. 174–184, Lecture Notes in Computer Science **780**, Springer-Verlag (1993).

[17] Harrison, J., *Theorem Proving with the Real Numbers*, Springer-Verlag, Berlin/Heidelberg/New York (1998).

[18] G. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science* **1** 27–52, 1975.

[19] McCune, W.: Otter 2.0, in: Stickel, M. E. (ed.), *10th International Conference on Automated Deduction* pp. 663–664, Springer-Verlag, Berlin/Heidelberg (1990).

[20] D. Miller and G. Nadathur. An Overview of λ-Prolog In *Proceedings of the Fifth International Symposium on Logic Programming, Seattle, August 1988.*

[21] Owre, S., Rushby, J. M., Shankar, N., PVS: A Prototype Verification System, in: Kapur, D. (ed.), *Automated Deduction–CADE-11, 11th International Conference on Automated Deduction*, 748–752, LNCS **607**, Springer-Verlag (1992).

[22] Pietrzykowski, T., and Jensen, D., A complete mechanization of second order logic, *J. Assoc. Comp. Mach.* **20** (2) pp. 333-364, 1971.

[23] Pietrzykowski, T., and Jensen, D., A complete mechanization of ω-order type theory, *ASsoc. Comp. Math. Nat. Conf.* 1972, Vol. 1, 82–92.

[24] Quaife, A., *Automated Development of Fundamental Mathematical Theories, Automated Reasoning, Vol.2*, Kluwer Academic Publishers, Dordrecht (1992).

[25] Yuan Yu, Computer proofs in group theory, *J. Automated Reasoning* **6**(3) pp. 251–286, 1990.