Mathematical Induction in Otter Lambda

Michael Beeson*

August 24, 2006

Abstract

Otter-lambda is Otter modified by adding code to implement an algorithm for lambda unification. Otter is a resolution-based, clause-language first-order prover that accumulates deduced clauses and uses strategies to control the deduction and retention of clauses. This is the first time that such a first-order prover has been combined in one program with a unification algorithm capable of instantiating variables to lambda terms to assist in the deductions. The resulting prover has all the advantages of the proof-search algorithm of Otter (speed, variety of inference rules, excellent handling of equality) and also the power of lambda unification. We illustrate how these capabilities work well together by using Otter-lambda to find proofs by mathematical induction. Lambda unification instantiates the induction schema to find a useful instance of induction, and then Otter's first-order reasoning can be used to carry out the base case and induction step. If necessary, induction can be used for those, too. We present and discuss a variety of examples of inductive proofs found by Otter-lambda: some in pure Peano arithmetic; some in Peano arithmetic with defined predicates; some in theories combining algebra and the natural numbers; some involving algebraic simplification (used in the induction step) by simplification code from MathXpert; and some involving list induction instead of numerical induction. These examples demonstrate the feasibility and usefulness of adding lambda unification to a first-order prover.

Introduction

Our purpose in this paper is to demonstrate the capabilities of the theorem prover Otter-lambda, or Otter-λ, [7] in finding proofs by mathematical induction. Otter-λ combines a new algorithm, lambda unification, with the well-known resolution-based theorem prover Otter [12]. Results obtained with this combination go beyond what can be done with standard Otter, because lambda unification can be used to instantiate a schema of mathematical induction, automatically finding the instance needed for a particular proof.

*Research supported by NSF grant number CCR-0204362.

Second-order (or higher-order) unification has typically been used in typed systems; see [7] for a detailed comparison of Otter-λ with other theorem provers. These systems have, however, not been especially good at automating proofs by induction. An excellent survey of the state of the art in inductive theorem proving (in 2001) is Bundy's *Handbook* article [9].[1] The best inductive theorem provers are ACL2 [8] (the present-day incarnation of the Boyer-Moore theorem prover), RRL [11], and Bundy's Oyster-Clam system in Edinburgh [10]. These provers have used special techniques to identify the correct induction schema to use, to identify the correct induction variable, and to find a good generalization of the theorem to be proved, should the original attempt not succeed. One should also mention the early work of Aubin [2], [3].

Otter-λ uses lambda unification to find the correct instance of induction; in the process it selects the induction variable. This is a non-deterministic process and Otter-λ does have the ability to backtrack, trying different choices of the induction variable, by returning more than one lambda unifier. The user can specify (by parameters set in the input file) the maximum number of unifiers to return for a single lambda-unification. Some heuristics are used to return the "most interesting" unifiers first. All but one of the examples in this paper, however, can be solved by returning only one unifier. We demonstrate the success of this approach by showing the Otter-λ can work a wide variety of examples of proof by induction. These fall into several classes:

- *Pure Peano arithmetic.* Using mathematical induction and the definitions of successor, addition, and multiplication, Otter-λ can prove the basic properties of addition and multiplication. Some of these properties require a double induction—the induction step and base case need to proved in turn by induction. Otter-λ finds all these instances of induction automatically.

- *Inductive proofs in algebra and analysis.* In these proofs, some variables range over natural numbers and some over a ring, or a group, or the real numbers.

- *Inductive proofs involving algebraic simplification.* Using a link to the computer algebra system of MathXpert, Otter-λ is able to find proofs by induction in which the induction step and/or base case requires considerable algebraic simplification.

- *List induction.* Otter-λ can work induction problems in other domains than the natural numbers.

We do not regard Otter-λ as a "combination of first-order logic and higher-order logic". Lambda logic is not higher-order, it is untyped. Lambda unification is not higher-order unification, it is unification in (untyped) lambda logic.

[1] The survey in [9] is now five years old, and of course research in inductive theorem-proving has continued; but the developments are not directly relevant to this paper, since we are taking a different approach to the subject, and the basic approaches described in [9] have not changed.

While there probably are interesting connections to typed logics, some of the questions about those relationships are open at present, and out of the scope of this paper. Similarly, while there are projects aimed at combining first-order provers and higher-order provers, that approach is quite different from ours. Otter-λ is a single, integrated prover, not a combination of a first-order prover and a higher-order prover. There is just one database of deduced clauses on which inferences are performed; there is no need to pass data between provers.

In this paper, we present a high-level, but precise and thorough, description of the lambda unification algorithm, and a number of examples of inductive proofs found using Otter-λ, illustrating the variety of induction problems that can be solved with Otter-λ, and discussing some of the difficulties involved. Proofs output by the prover have been included, since the discussion requires a detailed examination of those proofs. The complete input and output files are not included, but these files, and the prover itself, are available at the Otter-λ website [7].

Lambda logic and lambda unification

Lambda logic is the logical system one obtains by adding lambda calculus to first order logic. This system was formulated, and some fundamental metatheorems were proved, in an earlier publication [4]. The appropriate generalization of unification to lambda logic is this notion: two terms are said to be *lambda unified* by substitution σ if $t\sigma = s\sigma$ is provable in lambda logic. *Lambda unification* is an algorithm for producing lambda unifying substitutions. In Otter-λ, lambda unification is used, instead of only first-order unification, in the inference rules of resolution, factoring, paramodulation, and demodulation.

In Otter-λ input files, we write $lambda(x, t)$ for $\lambda x.\, t$, and we write $Ap(t, s)$ for t applied to s, which is often abbreviated in technical papers to $t(s)$ or even ts. In this paper, Ap and $lambda$ will always be written explicitly, so that we do not have to switch notation between the paper and the input files or the computer-produced proofs.

Although the lambda unification algorithm has been described in [4], we will describe it again here, since the reader must understand lambda-unification to follow the examples in this paper. As we define it here, lambda unification is a non-deterministic algorithm: it can return, in general, many different unifying substitutions for two given input terms. The input to the lambda-unification algorithm, like the input to ordinary unification, is two terms t and s (this time terms of lambda logic). The output, if the algorithm succeeds, is a substitution σ such that $t\sigma = s\sigma$ is provable in lambda logic.

We first give the relatively simple clauses in the definition. These have to do with first-order unification, alpha-conversion, and beta-reduction. The rule related to first-order unification just says that we try that first; for example $Ap(x, y)$ unifies with $Ap(a, b)$ directly in a first-order way. However, the usual recursive calls in first-order unification now become recursive calls to lambda unification. In other words: to unify $f(t_1, \ldots, t_n)$ with $g(s_1, \ldots, s_m)$ (according

to this rule) we must have $f = g$ and $n = m$; in that case we do the following:

```
σ =identity substitution;
for i = 1 to n {
    τ = unify(t_i, s_i);
    if (τ = failure)
        return failure;
    σ = σ ∘ τ; }
return σ;
```

Here the call to `unify` is a recursive call to the algorithm being defined. Since the algorithm is non-deterministic, there are choices to be made for each argument. For example, if there are two substitutions σ_i that unify a and c, and two ways to unify $b\sigma_i$ and $d\sigma_i$, then there will be four ways to unify $f(a, b)$ with $f(c, d)$.

To unify a variable x with a term t, return the substitution $x := t$ if t is identical to x or x is not bound and x does not occur in t.

The rule related to alpha-conversion says that, if we want to unify $lambda(z, t)$ with $lambda(x, s)$, first rename bound variables if necessary to ensure that x does not occur in t and z does not occur in s. Then let τ be the substitution $z := x$ and unify $t\tau$ with s, rejecting any substitution that assigns a value to x or a value depending on x.[2] If this unification succeeds with substitution σ, return σ.

The rule related to beta-reduction says that, to unify $Ap(lambda(z, s), q)$ with t, we first beta-reduce and then unify. That is, we unify $s[z := q]$ with t and return the result.

Lambda unification's most interesting instructions tell how to unify $Ap(x, w)$ with a term t, where t may contain the variable x, and t does not have main symbol Ap. Note that the occurs check of first-order unification does not apply in this case. The term w, however, is not allowed to contain x. In this case lambda unification is given by the following non-deterministic algorithm:

1. Pick a *masking subterm* q of t. That means a subterm q such that every occurrence of x in t is contained in some occurrence of q in t. (So q "masks" the occurrences of x; if there are no occurrences of x in t, then q can be any subterm of t, but see the next step.)

2. Call lambda unification to unify w with q. Let σ be the resulting substitution. If this unification fails, or assigns any value other than a variable to x, return failure. If it assigns a variable to x, say $x := y$ reverse the assignment to $y := x$ so that x remains unassigned.

3. If $q\sigma$ occurs more than once in $t\sigma$, then pick a set S of its occurrences. If q contains x then S must be the set of *all* occurrences of $q\sigma$ in t. Let z be a fresh variable and let r be the result of substituting z in $t\sigma$ for each occurrence of $q\sigma$ in the set S.

4. Append the substitution $x := \lambda z.\, r$ to σ and return the result.

[2] Care is called for in this clause, as illustrated by the following example: Unify $lambda(x, y)$ with $lambda(x, f(x))$. The "solution" $y = f(x)$ is wrong, since substituting $y = f(x)$ in $lambda(x, y)$ gives $lambda(z, f(x))$, because the bound variable is renamed to avoid capture.

There are two sources of non-determinism in the above, namely in steps 1 and 3. Otter-λ has a parameter `max_unifiers`, that can be set in the input file by a command like `assign(max_unifiers,9)`. In that case, lambda unification will backtrack over different selections of a masking subterm and set S, up to the maximum number of unifiers specified (per lambda unification). The default value of this parameter is one, in which case there is no backtracking, i.e. a deterministic selection is made. Even if backtracking is allowed, Otter-λ still attempts to pick "good" masking subterms according to some heuristics. Here are some of the heuristics used: in step 1, if x occurs in t, we prefer the smallest masking subterm q that occurs as a second argument of Ap.[3] If x occurs in t, but no masking subterm occurs as a second argument of Ap, we prefer the smallest masking subterm[4] If x does not occur in t, we pick a constant that occurs in t, or more generally a constant subterm of t; if there is none, we fail. Which constant subterm we pick is determined by some heuristics that seem to work well in the examples we have tried. In step 3, if q does not contain x, then an important application of this choice is to proofs by mathematical induction, where the choice of q corresponds to choosing a constant n, replacing some of the occurrences of n by a variable, and deciding to prove the theorem by induction on that variable. Therefore the choice of S is determined by heuristics that prove useful in this case. In particular, when proving equations by induction, we pick a constant that occurs on both sides of the equation, but not necessarily when proving non-equations. If there is a constant term of weight 1 that occurs on both sides of the equation, that term is used instead of a constant—this allows Otter-λ to "generalize" a goal, and since weight templates can be specified in the input file, it also gives the user some control over what terms can be selected as masking subterms. Our present heuristics call for never choosing a term of weight greater than 1; but weights can be set by the user in the input file, if it should be necessary.

Finally, lambda unification needs some rules for unifying $Ap(r, w)$ with t, when r is not a variable. The rule is this: create a fresh variable X, unify $Ap(X, w)$ with t generating substitution σ, then unify $X\sigma$ with $r\sigma$, generating substitution τ; if this succeeds return $\sigma\tau$, or rather, the substitution that agrees with $\sigma\tau$ but is not defined on X, since X does not occur in the original unification problem.

Example. Unify $Ap(Ap(x, y), z)$ with 3. Choose fresh X, unify $Ap(X, z)$ with 3, getting $z := 3$ and $X := lambda(u, u)$. Now unify $lambda(u, u)$ with $Ap(x, y)$, getting $y := lambda(u, u)$ and $x := lambda(v, v)$. So the final answer is $x := lambda(v, v)$, $y := lambda(u, u)$, $z := 3$. We can check that this really is a correct lambda unifier as follows:

$$Ap(Ap(x, y), z) \quad = \quad Ap(Ap(lambda(u, u), lambda(v, v)), 3)$$

[3]The point of this choice is that, if we want the proof to be implicitly typable, then q should be chosen to have the same type as w, and w is a second argument of Ap.

[4]This will not be done if the input file contains `set(types)`, because it might result in mis-typings; unless, of course, the input file also provides a `list(types)` that can be used to check the type of the masking subterm.

$$
\begin{aligned}
&= Ap(lambda(v, v), 3) \\
&= 3.
\end{aligned}
$$

Formulating induction in lambda logic

In this section, we show how mathematical induction is formulated in lambda logic, and how lambda unification can be used to instantiate the induction schema to produce a specific instance of induction needed for a particular proof. We begin by translating the usual first-order form of induction to clausal form. In writing this axiom, we use variables n, m to range over nonnegative integers, and $s(n)$ is the successor of n, that is, $n+1$; but in formal arithmetic, s is taken as primitive, and $+$ is defined.

Here is one common form of Peano's induction axiom, formulated with a variable X for sets of natural numbers:

$$0 \in X \wedge \forall n(n \in X \rightarrow s(n) \in X) \rightarrow \forall m(m \in X)$$

The theory logicians call "Peano Arithmetic" (PA) is a theory with variables only for numbers, not for sets, and the single induction axiom is replaced by an axiom schema, that is, by infinitely many axioms all of the same form, obtained by replacing "$n \in X$" by $P(n)$, for any formula P with one free variable:

$$P(0) \wedge \forall n(P(n) \rightarrow P(s(n))) \rightarrow \forall m(P(m)).$$

To formulate induction in lambda logic, we replace "$n \in X$" by $Ap(X, n)$. When $Ap(X, n)$ occurs in the syntactic position of a formula, it is synonymous with $Ap(X, n) = true$; now X ranges over propositional functions (boolean-valued functions, in today's terminology) defined on the natural numbers. We get the following formulation:[5]

$$Ap(X, 0) \wedge \forall n(Ap(X, n) \rightarrow Ap(X, s(n))) \rightarrow \forall m(Ap(X, m)). \tag{1}$$

Lambda logic is an untyped theory, but the version of induction we have just given is implicitly typed: the variables n and m have type N (the type of natural numbers), and the variable X has type $i(N, bool)$ (the type of functions from N to $bool$). In turn Ap has type $i(i(N, bool), bool)$; s has type $i(N, N)$; the constant 0 has type N. One might think that we should introduce a unary predicate $N(x)$ and restrict the integer variables to $N(x)$ to achieve a proper formalization of arithmetic in lambda logic, but this approach, as in first-order many-sorted logic, leads to inefficiencies in automated deduction (notably, it interferes with the applicability of hyperresolution). We therefore proceed without such explicit typings. In [5], we have proved a theorem justifying this procedure: if the axioms

[5]This theory does not express the full strength of Peano's informal axioms, since there are many predicates on natural numbers that cannot be defined in this theory. The situation is more like what logicians call "weak second order arithmetic", which has variables for sets, and induction is expressed using a single formula with a set variable, but the theory is still weak because it can't prove many sets exist.

can be correctly typed (in a certain precise sense), and Otter-λ finds a proof, then all the steps of the proof can be correctly typed as well. The typings we just gave for s, 0, and Ap are such that the induction axiom is correctly typed, so [5] justifies us in omitting a unary predicate $N(x)$ in the formulation of induction.

Our next step is to convert (1) to clausal form. Using the equivalence of $A \to B$ with $\neg A \vee B$, we get

$$\neg Ap(X,0) \vee \exists n(Ap(X,n) \wedge \neg Ap(X,s(n))) \vee \forall m(Ap(X,m)).$$

Now we must introduce a Skolem function g and replace n by $g(X)$. We get

$$\neg Ap(X,0) \vee (Ap(X,g(X)) \wedge \neg Ap(X,s(g(X)))) \vee \forall m(Ap(X,m)).$$

Distributing the $\wedge$ over $\vee$ we arrive at the clausal form. In clausal form it is traditional to use '$-$' for negation, instead of '$\neg$', and to use | for disjunction instead of $\vee$. We obtain the following two clauses as the clausal form of mathematical induction:

$$-Ap(X,0) \quad | \quad Ap(X,g(X)) \quad | \quad Ap(X,w) \tag{2}$$
$$-Ap(X,0) \quad | \quad -Ap(X,s(g(X))) \quad | \quad Ap(X,w) \tag{3}$$

In all clausal-form formulas, we follow the convention that variables have names beginning with (upper or lower case) x, y, z, u, v or w and other letters are constants. To follow this convention, we changed the quantified variable m to w in passing to clausal form, where m would be considered constant. Note that if we had proceeded in another order, we might have come out with $g(X,w)$ instead of just $g(X)$; that would also work, but it is simpler to Skolemize before removing the quantifier on m. Clausal form is not unique.

This is one form in which induction can written in an Otter-λ input file; specifically, this is the form we use when proving theorems in Peano Arithmetic. We will show below that Otter-λ can prove the standard theorems at the foundation of arithmetic—for example, the associativity and commutativity of addition and multiplication, and the distributive laws—directly from the definitions of addition, multiplication, and successor, using this formulation of induction and lambda unification. Other forms of induction will also be introduced and used in other examples below, but we will begin with this one, the archetypal form. Peano arithmetic also includes the usual axioms for successor:

$$s(x) \neq 0 \tag{4}$$
$$s(x) \neq s(y) \quad | \quad x = y. \tag{5}$$

Lambda unification applied to proof by induction

In this section, we will show how lambda unification works to find an instance of induction. Let us pick a sample problem: the associativity of addition. Our

axioms will include the definition of addition, namely

$$\begin{aligned} x + 0 &= x && (6)\\ x + s(y) &= s(x + y) && (7)\\ & && (8) \end{aligned}$$

To prove the associativity of addition, we enter its negation as an axiom, with the variables changed to constants, as usual in clausal proof systems:

$$(a + b) + c \neq a + (b + c) \tag{9}$$

We now show how binary resolution, extended to use lambda unification instead of just ordinary (Robinson) unification, can be used in this axiom set. We resolve the negated goal with (2) and/or with (3). Either of these attempted resolutions gives rise to the unification problem, to unify $Ap(X, w)$ with $(a + b) + c = a + (b + c)$. We follow the steps for lambda unification given above. Step 1 requires us to pick a masking term. Since there are no occurrences of X in the associative law, we could pick any term. But Otter-λ's algorithm will pick either a constant or a term occurring as a second argument of Ap. There are no occurrences of Ap, so we have to pick one of the constants a, b, or c. Let's pick c. We then unify w with c, getting the substitution σ that assigns w the value c. Continuing to step 3 of lambda unification, we have to pick a set S of occurrences of $c\sigma$ (which is just c in this case) in the associative law. Let's pick the set S consisting of both occurrences of c. Then the instructions for lambda unification say, "let z be a fresh variable, and let r be the result of substituting z for each occurrence of c". The term r that we get in this way is $(a + b) + z = a + (b + z)$. Step 4 now tells us to produce the substitution $X := lambda(z, (a + b) + z = a + (b + z)), w := c$ as the result of lambda unification. Let's verify that in this case, we have indeed produced a lambda unifier. Call this substitution τ. Applying τ to $Ap(X, w)$, we find

$$\begin{aligned} Ap(X, w)\tau &= Ap(lambda(z, (a + b) + z = a + (b + z)), c)\\ &= (a + b) + c = a + (b + c) \end{aligned}$$

so indeed, τ is a lambda unifier of the two input terms. This value of X is exactly the instance of induction required to prove the associative law. We now show how the proof proceeds. By resolving the negated associative law with $Ap(X, w)$ in (2) we derive

$$-Ap(X\tau, 0) \quad | \quad Ap(X\tau, g(X\tau))$$

If we write out $X\tau = lambda(z, (a+b)+z = a+(b+z))$ explicitly, but abbreviate the Skolem term $g(X\tau)$ by d, this becomes

$$\begin{aligned} -Ap(lambda(z, (a + b) + z = a + (b + z)), 0) \quad |\\ Ap(lambda(z, (a + b) + z = a + (b + z)), d) \end{aligned}$$

Beta-reducing, we have

$$(a+b)+0 \neq a+(b+0) \quad | \quad (a+b)+d = a+(b+d).$$

Using the law (6), the first literal reduces to $a+b \neq a+b$. This resolves with the equality axiom $x = x$ and falls away, leaving

$$(a+b)+d = a+(b+d),$$

which is recognizable as the induction hypothesis. We now start again, resolving the negated associative law with $Ap(X,w)$ in (3). The same lambda unification problem arises, and the same substitution is produced. We obtain this time

$$\begin{array}{l} -Ap(lambda(z,(a+b)+z = a+(b+z)),0) \quad | \\ -Ap(lambda(z,(a+b)+z = a+(b+z)),s(d)) \end{array}$$

and after beta-reduction,

$$(a+b)+0 \neq a+(b+0) \quad | \quad (a+b)+s(d) \neq a+(b+s(d)).$$

Again using (6) to dispose of the base case (the first literal) we obtain

$$(a+b)+s(d) \neq a+(b+s(d)),$$

which is recognizable as the negated induction step. Using the second law of addition (7), we obtain

$$s((a+b)+d) \neq a+s(b+d))$$

and then using it again on the right we have

$$s((a+b)+d) \neq s(a+(b+d)).$$

Now one application of the induction hypothesis yields

$$s(a+(b+d)) \neq s(a+(b+d)),$$

which resolves with the equality axiom $x = x$, producing the empty clause and completing the proof.

This has been a hand-produced proof by binary resolution and equality reasoning, using lambda unification in resolution. We now discuss how Otter-λ finds this proof. We will assume familiarity with the basics of the clausal-search paradigm of automated deduction, including the rules of inference binary resolution, hyperresolution, paramodulation, and demodulation, all of which are explained (for example) by Wos and Pieper [15]. We also assume familiarity with the terms "set of support" and "usable". These terms are also defined in [15]; on page 94 of [15] the basic search algorithm of Otter, which involves lists of formulas with these names, is lucidly explained.

We begin by preparing an input file: we put the negated goal in the set of support, and the other axioms in "usable". We include the equality axiom $x = x$; in the context of lambda logic, this says that $Ap(x, y)$ is always defined, i.e., we are working with (total) lambda logic rather than partial lambda logic. We put the two axioms about addition in as demodulators, oriented so that $x + s(y)$ will be rewritten as $s(x + y)$ and $x + 0$ will be changed to x. Otter-λ applies beta-reduction in the same way that Otter applies demodulation. Thus several steps at the last of the hand crafted proof above actually get compressed into one Otter-λ step, as beta reductions and demodulations are applied as "simplifications" of deduced clauses, and the intermediate steps shown above are not retained as deduced clauses. The proof that Otter-λ produces is thus somewhat shorter in outward appearance, although in reality it represents the same proof shown above. Otter-λ proofs follow the same format as Otter proofs; those not accustomed to reading Otter proofs will find some hints below. Here is the proof exactly as Otter-λ produces it, except that extra line breaks have been inserted to make it print within the margins:

```
1 [] x+0=x.
3 [] x=x.
6 [] -ap(y,0)|ap(y,g(y))|ap(y,z).
7 [] -ap(y,0)| -ap(y,s(g(y)))|ap(y,z).
8 [] x+s(y)=s(x+y).
10 [] (a+b)+n!=a+b+n.
11 [binary,10.1,7.3,demod,beta,1,1,beta,unit_del,3]
   (a+b)+s(g(lambda(x,(a+b)+x=a+b+x)))!=
   a+b+s(g(lambda(x,(a+b)+x=a+b+x))).
12 [binary,10.1,6.3,demod,beta,1,1,beta,unit_del,3]
   (a+b)+g(lambda(x,(a+b)+x=a+b+x))=a+b+g(lambda(x,(a+b)+x=a+b+x)).
14 [para_from,12.1.1,8.1.2.1]
   (a+b)+s(g(lambda(x,(a+b)+x=a+b+x)))=
   s(a+b+g(lambda(x,(a+b)+x=a+b+x))).
18 [para_into,11.1.2.2,8.1.1]
   (a+b)+s(g(lambda(x,(a+b)+x=a+b+x)))!=
   a+s(b+g(lambda(x,(a+b)+x=a+b+x))).
25 [para_into,14.1.2,8.1.2]
   (a+b)+s(g(lambda(x,(a+b)+x=a+b+x)))=
   a+s(b+g(lambda(x,(a+b)+x=a+b+x))).
26 [binary,25.1,18.1] $F.
```

and here it is again, but with the Skolem term `g(lambda(x,(a+b)+x=a+b+x))` replaced by a constant `d`; this makes the proof much easier to read. In general such a Skolem term essentially represents an "arbitrary constant".

```
1 [] x+0=x.
3 [] x=x.
6 [] -ap(y,0)|ap(y,g(y))|ap(y,z).
7 [] -ap(y,0)| -ap(y,s(g(y)))|ap(y,z).
```

```
8 [] x+s(y)=s(x+y).
10 [] (a+b)+n!=a+b+n.
11 [binary,10.1,7.3,demod,beta,1,1,beta,unit_del,3]
   (a+b)+s(d)!=a+b+s(d).
12 [binary,10.1,6.3,demod,beta,1,1,beta,unit_del,3]
   (a+b)+d=a+b+d.
14 [para_from,12.1.1,8.1.2.1] (a+b)+s(d)=s(a+b+d).
18 [para_into,11.1.2.2,8.1.1] (a+b)+s(d)!=a+s(b+d).
25 [para_into,14.1.2,8.1.2] (a+b)+s(d)=a+s(b+d).
26 [binary,25.1,18.1] $F.
```

Hints for reading Otter proofs: The numbers at the left are line numbers. They indicate the number of each deduced clause among all clauses generated during the search. If some of the numbers are large, that means that a lot of clauses were generated. After the numbers on each line come some square brackets. If there is nothing inside these brackets, that means that this line was an axiom. If there is something inside, that tells what rule or rules of inference were used to deduce this line, and the numbers tell what the "parent clauses" were, i.e., from which previous lines this line was deduced. For example, in the above proof, line 11 was deduced by binary resolution from lines 10 and 7, but the direct result of resolution was simplified by demodulation, beta reduction, and unit deletion. The extra digits after the decimal points tell which part of the formula was used, e.g. 10.1 is the first literal in line 10, and 7.3 is the third literal in line 7. When line 14 is deduced, the numbers 12.1.1 refer to the first subterm of the first literal of line 12, that is, to the left side of that equation. If you are new to reading Otter proofs, compare the above example in detail with the hand crafted proof, until you see what each of the annotations in square brackets means.

Choosing the right induction variable

The above example illustrates the problem of choosing an induction variable. When lambda unification has to pick a masking subterm of $(a{+}b){+}c = a{+}(b{+}c)$, there are nine possible choices to consider (all subterms of the two sides). Otter-lambda can backtrack and return multiple unifiers, but this improvement to the implementation was made rather late in this research(December 2005), after a long hesitation about exactly how to do it, since Otter's architecture seemed at first to depend heavily on the single-valuedness of unification. Therefore, attention was paid to various heuristics for making a good selection; these heuristics are still used if several selections are allowed, but they are no longer necessary. Otter-λ allows a user to put a command of the form `assign(max_unifiers, 8)` in an input file. This causes Otter-λ to backtrack over different selections of masking terms and sets of occurrences of the masking term, up to the specified maximum number (per unification of $Ap(X, w)$ with t) and return multiple unifiers corresponding to these choices.

In choosing a masking term to be replaced by an induction variable, it is also important to choose a term that is "implicitly typed" as an integer, in the sense of [5]. This is necessary if we want to be assured in advance by the theorems of [5] that our deductions will be correctly typeable. This can be assured by putting `set(types)` in the input file, in which case only constants will be tried; unless the input file also contains a `list(types)`, in which the types of some other terms can be specified. The heuristic used to select a constant, when several occur, is this: if the main symbol is '=', and some constant occurs on both sides of the equation, then select the rightmost constant that occurs on both sides of the equation. Otherwise (if the symbol is not equality or no constant occurs on both sides) select the rightmost constant. However, if one of the constants is literally '`n`' or '`m`', then select that constant—this feature allows the writer of the input file to give Otter-λ a hint.

Such hints were important in early versions of Otter-lambda, which could not backtrack to produce multiple unifiers, but they are no longer necessary. All the examples in this paper, except those involving the need to generalize the theorem before proving it by induction, can be proved by Otter-λ without needing to backtrack over different choices of the induction variable, but if backtracking is allowed, the method is essentially free of the need for heuristics; we could take those heuristics out of the program without diminishing its power.

Peano arithmetic

We have already seen one simple example of a proof in Peano arithmetic, but Otter-lambda has proved other, more complicated examples. Each of these examples has some points of interest, so we review them here.

The cancellation law $a + n = b + n$ *implies* $a = b$. Since what is to proved is an implication, the input file contains the assumption $a + n = b + n$ and the negated conclusion $a \neq b$.

Otter-λ then proves $a + n \neq b + n$ by induction, using $a \neq b$ for the base case. Since the theorem being proved by induction is an inequation rather than an equation, induction needs to be given in a slightly different form, with Ap and $-Ap$ interchanged. It takes both forms to fully express induction in lambda logic, since the meta-level negation is not expressed at the object level. In other words, Otter-λ is never going to try to unify $Ap(y, x)$ with $-P(x)$, so if we want induction to apply to negated literals, we have to supply also the form in which $Ap(y, x)$ is negated.

Commutativity of addition

This example is of interest because the induction step requires a lemma, and the lemma itself has to be proved by induction. Not only that, the base case also has to be proved by induction. This is an inevitable situation in inductive theorem-proving, as is pointed out in Bundy's survey article [9], p. 869:

> Sometimes a lemma required to complete the proof is not already available and is not deducible from the existing theory without a nested application of induction. This is a consequence of the failure of cut elimination for inductive theories. Such lemmata must be conjectured and then proved as subgoals.

Here is an outline of the proof: we want to prove $x+y=y+x$ by induction on y. The base case is $x+0=0+x$. Since $x+0=x$ is one of the Peano axioms, this boils down to $x=0+x$. That has to be proved by induction on x. Putting that aside for now, the induction step of the main induction is to prove

$$x+s(y)=s(y)+x$$

assuming $x+y=y+x$. This lemma also has to be proved by induction. Thus three applications of the induction axiom are needed to complete this proof. Otter-lambda successfully finds all three instances of induction automatically, using lambda unification. In fact, Otter-lambda proves $s(y)+x=s(x+y)$ by induction, but that is only one step removed (by the definition of addition) from the induction step of the main induction.

Now we examine the proof more closely. The first thing to notice is the three lambda terms that occur in the proof:

```
lambda(x,a+x=x+a)
lambda(y,y=0+y)
lambda(y,s(g(n,lambda(z,a+z=z+a))+y)=s(g(n,lambda(u,a+u=u+a)))+y)
```

These terms represent the propositional functions to be used in the application of the induction schema. The corresponding Skolem terms are

```
c = g(n,lambda(x,a+x=x+a))
d = g(a,lambda(y,y=0+y))
b = g(a,lambda(y,s(g(n,lambda(z,a+z=z+a))+y)=
    s(g(n,lambda(u,a+u=u+a)))+y))
```

The following version of the proof was obtained from the machine output by replacing these Skolem terms (and variants different only by renaming of lambda-bound variables) by `c`, `d`, and `b` as defined here. I have also stripped off the names of the inference rules used, leaving only the line numbers of the parents involved.

```
1 [] x+0=x.
2 [] x=x.
5 [] -ap(y,0)|ap(y,g(z,y))|ap(y,z).
6 [] -ap(y,0)| -ap(y,s(g(z,y)))|ap(y,z).
7 [] x+s(y)=s(x+y).
8 [] a+n!=n+a.
```

```
9 [8.1,6.3] a!=0+a|a+s(c)!=s(c)+a.
10 [8.1,5.3] a!=0+a|a+c=c+a.
27 [9.1,6.3,1,2] a+s(c)!=s(c)+a|s(d)!=0+s(d).
28 [9.1,5.3,1,2] a+s(c)!=s(c)+a|d=0+d.
51 [10.1,6.3,1,2] a+c=c+a|s(d)!=0+s(d).
52 [10.1,5.3,1,2] a+c=c+a|d=0+d.
85 [52.2.2,7.1.2.1] 0+s(d)=s(d)|a+c=c+a.
150 [28.2.2,7.1.2.1] 0+s(d)=s(d)|a+s(c)!=s(c)+a.
962 [85.1.1,51.2.2] a+c=c+a.
968 [962.1.1,7.1.2.1] a+s(c)=s(c+a).
976 [968.1.1,7.1.1] s(c+a)=s(a+c).
991 [976.1.2,7.1.2] s(c+a)=a+s(c).
2852 [150.1.1,27.2.2,2] a+s(c)!=s(c)+a.
2873 [2852.1.1,991.1.2] s(c+a)!=s(c)+a.
2880 [2873.1,6.3,1,1,2] s(c+s(b))!=s(c)+s(b).
2881 [2873.1,5.3,1,1,,2] s(c+b)=s(c)+b.
2893 [2881.1.1,7.1.2] c+s(b)=s(c)+b.
2927 [2893.1.1,7.1.2.1] c+s(s(b))=s(s(c)+b).
3015 [2880.1.2,7.1.1] s(c+s(b))!=s(s(c)+b).
3091 [2927.1.1,7.1.1] s(c+s(b))=s(s(c)+b).
3092 [3091.1,3015.1] $F.
```

Now for the commentary. Line 8 is the negation of the main goal $a+n = n+a$. Lines 9 and 10 say, Let's prove it by induction on n. Lines 27, 28, 51, and 52 say, OK, the base case boils down to $0 + a = a$; let's prove that by induction on a. (It needs four lines to say that.) The next three lines, 85, 150, and 962, polish off that induction, like this: assuming $0 + d = d$, take the successor of both sides to get $s(0 + d) = 0 + s(d) = s(d)$. The base case is taken care of by the demodulator $x + 0 = 0$. At line 962 the inductive proof of the base case of the main induction is completed, leaving the now-unencumbered induction hypothesis of the main induction, $a + c = c + a$. Again taking the successor of both sides we have $s(a + c) = a + s(c) = s(c + a)$ (lines 968, 976) and using the definition of addition on $s(c + a) = s(a + c)$, we get $s(c + a) = a + s(c)$ (line 991). Some eighteen hundred clauses later, we find the negated goal of the main induction step at clause 2852: $a + s(c) \neq s(c) + a$. Otter-lambda applies the definition of addition (backwards!) to the left hand side to get, at line 2873, $s(c + a) \neq s(c) + a$.

Then lines 2880 and 2881 say "Let's prove that by induction on a." Otter-lambda chooses the constant a because it programmed to prefer a constant over a Skolem term for replacement by a new lambda variable in unification—remember c is a complicated Skolem term—even when that Skolem term occurs as a second argument of Ap. The induction hypothesis is $s(c + b) = s(c) + b$ (line 2881); that can be written $c + s(b) = s(c) + b$, and taking the successor of both sides we have $s(c + s(b)) = s(s(c) + b)$ (line 2927) and hence, using the definition of addition on the left, $s(c+s(b)) = s(s(c)+b)$ (line 3091). Using the definition of addition once more on the right side, we get $s(c+s(b)) = s(c)+s(b)$.

That completes the induction step and the proof. Actually, Otter-lambda does this last step slightly differently: it applies the definition of addition to the negated goal of the induction step, which is $s(c + s(b)) \neq s(c) + s(b)$, getting $s(c + s(b)) \neq s(s(c) + b)$, contradicting line 3091.

Commutativity of multiplication

The commutativity of addition requires about four seconds on a 2.8 ghz. machine. Our next example, the commutativity of multiplication, requires about 42 seconds on the same machine. About 149,000 clauses are generated. The proof, however, is only 12 steps long, and is about what a human would do. You are recommended to find a proof yourself, using pencil and paper, before looking at the proof found by Otter-λ. The input file contains, in addition to the Peano axioms, some consequences of the Peano axioms that are proved in separate Otter-lambda runs from the Peano axioms: namely, the three examples discussed above, and the lemma $x + s(y) = s(x) + y$ proved as the induction step of the associativity of addition.

The first part of the proof lists the axioms that are used; we list that part here:

```
1 [] x+0=x.
2 [] x*0=0.
3 [] 0*x=0.
12 [] x=x.
15 [] -ap(y,0)|ap(y,g(z,y))|ap(y,z).
16 [] -ap(y,0)| -ap(y,s(g(z,y)))|ap(y,z).
18 [] x+y+z= (x+y)+z.
19 [] x+y=y+x.
20 [] x*s(y)=x*y+x.
21 [] x+s(y)=s(x)+y.
23 [] m*n!=n*m.
```

The way to approach an Otter-λ proof by induction is to first look at the λ-terms. They tell you what Otter-λ has decided to prove by induction. In this case the λ-term corresponding to the main induction is $lambda(x, x * n = n * x)$, which says that Otter-λ is going to prove $x * n = n * x$ by induction on x. Then, using a text editor, replace all occurrences of the Skolem term $g(lambda(x, x * n = n * x))$ (and terms that differ from it only by renaming the bound variable x) by a constant c. That exposes the second induction in this proof, in which Otter-lambda proves $s(c) * y = c * y + y$ by induction on y. Now, using a text editor, replace the Skolem term $g(n, lambda(y, s(c) * y = c * y + y))$ by a constant b. Finally, to make the proof fit nicely on the printed page, we omit the names of the inference rules used, leaving only the line numbers of the parents to indicate the deduction steps. For example, the first deduced step is shortened from

```
24 [binary,23.1,16.3,demod,beta,3,2,beta,unit_del,12]
```

```
s(c)*n!=n*s(c).
```

to simply

```
24 [23,16,3,2,12] s(c)*n!=n*s(c).
```

Here are the deduced steps of the proof:

```
24 [23,16,3,2,12] s(c)*n!=n*s(c).
25 [23,15,3,21,12] c*n=n*c.
28 [25,20] n*s(c)=c*n+n.
54 [28,24] s(c)*n!=c*n+n.
125 [54,16,2,2,1,12] s(c)*s(b)!=c*s(b)+s(b).
126 [54,15,2,2,1,12] s(c)*b=c*b+b.
230 [126,20] s(c)*s(b)= (c*b+b)+s(c).
877 [125,20] s(c)*s(b)!= (c*b+c)+s(b).
1773 [230,18] s(c)*s(b)=c*b+b+s(c).
21899 [877,18] s(c)*s(b)!=c*b+c+s(b).
23692 [1773,19] s(c)*s(b)=c*b+s(c)+b.
89895 [21899,21] s(c)*s(b)!=c*b+s(c)+b.
89896 [89895,23692] $F.
```

Let us look at this line by line. The first two lines are the induction step to be proved. The base case has already been dealt with, which makes sense since $x*0 = 0$ is one of Peano's axioms and we also gave it $0*x = 0$. The next two lines apply the definition of multiplication, and as already remarked, Otter-lambda then "decides" to prove 28 by induction on n. Lines 125 and 126 record this "intention", stating the induction step to be proved. What happened to the base case? That would be $0 * s(c) = c * 0 + 0$. Both sides have demodulated to zero and unit deletion has removed that literal. Line 230 appears to be a clever step: make the left side of 126 match the left side of 125 by adding $s(c)$ to both sides and then using the definition of multiplication backwards to collapse the left side. Of course, Otter-λ has no such "intention"—it just uses paramodulation when it can, and this clause turned out to be useful. 877 results from using the definition of multiplication on 125, the negated goal of the induction step. At this point a human can see the proof coming: we just need to manipulate the right-hand sides of 230 and 877 into the same form. The next lines apply associativity of addition and the given fact that $b + s(c) = s(b) + c$ to complete the proof.

Why did it take more than sixty thousand clauses (with the original input file) to find the last step of this proof? Because it took that long for 21899 to become the given clause. That clause has weight 14, and there are lot of clauses of that weight or smaller, but part of the problem is that the input file had, for this run, `pick_given_ratio` set to 4, so that every fifth given clause had weight 20 or so, and these heavy clauses generated a lot of lower weight clauses that got in the way. Observing that, and observing that no clauses heavier than 14

are required, I changed the input file, removing `pick_given_ratio`, and setting max_weight to 14. This cut the running time from 17 minutes to less than one minute, and decreased the number of generated clauses by about 40%.

Transitivity of Order

Order is defined in PA by $x \leq y$ iff $\exists z(x + z = y)$. Lambda logic permits treating quantifiers as operations that apply to propositional functions, so that $\exists z Ap(y, z)$ is rendered as `exists(lambda(z,Ap(y,z)))`; hence this definition can be formulated directly in lambda logic, rather than requiring a Skolemization first, as would be required in a first-order prover. We show how Otter-λ is able to handle this kind of definition, and that it can prove the transitivity of equality so defined. Although induction is not required, some inductive proofs involving order are discussed below, so we need to develop the concept of order.

In lambda logic, the existential quantifier is represented by a constant `exists`, and the two "laws of existence" are as follows:

```
-Ap(Z,w) | exists(lambda(x, Ap(Z,x))).   % first law of existence
-exists(lambda(x,Ap(Z,x))) | Ap(Z, e(Z)).% second law of existence
```

The reader will recognize these laws as corresponding to the usual quantifier axioms in first-order logic: $e(Z)$ is similar to a "fresh" variable, i.e. one that does not occur free in the rest of the clause. Also, $e(Z)$ is similar to Hilbert's ϵ-symbol, which he wrote $\epsilon x.Z(x)$, and means "some x such that $Z(x)$, if there is one." In lambda logic, `exists` is applied to predicates, so $\exists x\ P(x)$ would become $Ap(\exists, \lambda x.\, P(x))$, or perhaps $Ap(\exists, \lambda x.\, Ap(P, x)$, if P is considered a constant instead of a predicate. (Both are possible in lambda logic.)

Now the definition of $n \leq m$ for integers n and m can be expressed this way:

```
-(x <= y) | exists(lambda(z,x +z= y)).
-exists(lambda(z,x+z = y)) | x <= y.
```

These formulae, together with Peano's axioms in the form used in the previous examples, and the associativity of addition, go in `list(usable)`. Then we give Otter-λ the goal of proving the transitivity of equality as follows:

```
list(sos).
a <= b.
b <= c.
-(a <= c).
end_of_list.
```

It turns out that induction is not needed for the proof, since we supplied the associativity of addition. Here is the proof it finds, which corresponds nicely to the natural proof:

```
8 [] -Ap(Z,w)|exists(lambda(x,Ap(Z,x))).
9 [] -exists(lambda(x,Ap(Z,x)))|Ap(Z,e(Z)).
11 [] -(x<=y)|exists(lambda(z,x+z=y)).
12 [] -exists(lambda(z,x+z=y))|x<=y.
13 [] (x+y)+z=x+y+z.
14 [] a<=b.
15 [] b<=c.
16 [] -(a<=c).
17 [binary,14.1,11.1] exists(lambda(x,a+x=b)).
19 [binary,15.1,11.1] exists(lambda(x,b+x=c)).
21 [binary,16.1,12.2] -exists(lambda(x,a+x=c)).
31 [binary,17.1,9.1,demod,beta] a+e(lambda(x,a+x=b))=b.
33 [binary,19.1,9.1,demod,beta] b+e(lambda(x,b+x=c))=c.
36 [binary,21.1,8.2,demod,beta] a+x!=c.
40 [para_into,36.1.1,13.1.2] (a+x)+y!=c.
367 [para_into,33.1.1.1,31.1.2]
    (a+e(lambda(x,a+x=b)))+e(lambda(y,b+y=c))=c.
368 [binary,367.1,40.1] $F.
```

Trichotomy of Order

Continuing with the theory of order, the next natural theorem to prove is the trichotomy law: it is contradictory to assume $a \leq b$ and $b \leq a$ and $a \neq b$. For this problem, the input file assumes (as well as induction and the definition of addition): the associativity of addition, the "laws of existence", and two more facts: if $x + y = x$ then $y = 0$, and if $x + y = 0$ then $y = 0$. These are expressed in the input file as follows, with line numbers from the proof for later reference:

```
18 x+y!=x | y=0.
20 x+y!=0 | y=0
```

Of course, these can be proved in turn by induction, but the point of this example is the correct manipulation of a definition that involves "there exists", so we just put those needed extra formulas in.

The proof, informally, goes like this. Since $a \leq b$, that means there exists an x such that $a+x = b$. Fix such an x, call it $e1$. Then $a+e1 = b$. Similarly, since $b \leq a$, there exists an x such that $b + x = a$. Fix such an x, call it $e2$. Then $b + e2 = a$. Therefore $(a + e1) + e2 = a$. By associativity, $a + (e1 + e2) = a$. Then by 18, $e1 + e2 = 0$. By 20, $e1 = 0$. Then $b = a + e1 = a + 0 = a$. That contradicts $a \neq b$, and completes the proof.

Here is the proof that Otter-λ finds (with the names of the inference rules removed):

```
 [] x+0=x.
2 [] (x+y)+z=x+y+z.
11 [] -exists(lambda(x,Ap(Z,x)))|Ap(Z,e(Z)).
```

```
16 [] -(x<=y)|exists(lambda(z,x+z=y)).
18 [] x+y!=x|y=0.
20 [] x+y!=0|y=0.
21 [] a<=b.
22 [] b<=a.
23 [] a!=b.
24 [21,16] exists(lambda(x,a+x=b)).
26 [22,16] exists(lambda(x,b+x=a)).
31 [24,11,demod,beta] a+e(lambda(x,a+x=b))=b.
32 [26,11,demod,beta] b+e(lambda(x,b+x=a))=a.
35 [31.1.1,18.1.1] b!=a|e(lambda(x,a+x=b))=0.
58 [32.1.1.1,31.1.2,demod,2]
       a+e(lambda(x,a+x=b))+e(lambda(y,b+y=a))=a.
66 [32.1.2,23.1.1] b+e(lambda(x,b+x=a))!=b.
91 [35.2.1,31.1.1.2,demod,1] a=b|b!=a.
94 [91.1.1,32.1.2] b+e(lambda(x,b+x=a))=b|b!=a.
275 [58,18] e(lambda(x,a+x=b))+e(lambda(y,b+y=a))=0.
820 [94,66] b!=a.
823 [820.1.1,31.1.2] a+e(lambda(x,a+x=b))!=a.
1308 [275,20] e(lambda(x,b+x=a))=0.
1321 [1308.1.1,58.1.1.2.2,demod,1] a+e(lambda(x,a+x=b))=a.
1322 [321.1,823.1] $F.
```

The proof uses the "second law of existence" to formalize the line "Fix such an x, call it $e1$." In the Otter-λ proof, instead of $e1$ it is called `e(lambda(x,a+x=b))`. The proof looks quite natural, but appearances are deceptive: it was not easy to get Otter-λ to find this proof. If I had cheated by leaving out the first law of existence, it would have been quite easy, but the first law of existence is there in the input file, even though it is not used in the proof. At first, it generated lots of useless conclusions, that swamped Otter-λ in a sea of nested `exists` and `lambda` terms, and prevented the proof from being found. Intuitively, it reasoned like this: Say it derived $2+2=4$. Well then, there exists an x such that $2+2=4$. And there exists a y such that there exists an x such that $2+2=4$. And there exists a z such that there exists a y such that there exists an x such that $2+2=4$. And so on, until `max_weight` is exceeded. But by that time, even with a relatively low `max_weight`, enough conclusions have been generated to clog up the works, specifically to clog up the set of support.

The solution to this difficulty is to tell Otter-λ to discard conclusions with nested `exists` or nested `lambda`. Luckily, Otter (and hence Otter-λ) has a mechanism for doing that (available since Otter version 3.0.3, April 1994). It uses the syntax `$dots` to give the unwanted nested terms a weight greater than `max_weight`, which will cause them to be discarded.

Why is it "cheating" to just omit the first law of existence from the input file, and not "cheating" to put in a directive to eliminate its unwanted consequences? Because the point of the example is to show how to work with existence and its laws using lambda unification plus standard first-order technques. A human

does not get sidetracked making useless deductions as illustrated, precisely because s/he can recognize them as useless. The directive in question tells Otter-λ that conclusions of a certain form are going to be useless. This kind of directive is a standard technique when using Otter, so we are simply showing that, it is all right to include the lambda axioms for "there exists", because standard techniques from first-order proving can be applied to control unwanted conclusions, also in this new setting.

More inductive proofs involving order

The next example is this theorem: $a \leq 0 \rightarrow a = 0$. Otter-$\lambda$ is able to automatically find an inductive proof. Here is a sketch of the proof: First, Otter-λ uses the "laws of existence", applied to $a \leq 0$, to deduce $a + e(lambda(x, a + x = 0)) = 0$. Then proves $a + n \neq 0$ by induction on n. The base case is the negated goal $a \neq 0$. For the induction step, if $a + s(n) = 0$, then by the definition of addition, $s(a + n) = 0$, contradicting one of Peano's axioms.

The proof produced by Otter-λ is quite succinct: the entire induction argument takes place in one heavily-annotated step, line 18 below:

```
1 [] a!=0.
3 [] s(x)!=0.
6 [] ap(y,0)|ap(y,s(g(y)))| -ap(y,z).
9 [] -exists(lambda(x,ap(Z,x)))|ap(Z,e(Z)).
11 [] x+0=x.
12 [] x+s(y)=s(x+y).
13 [] (u<=v)=exists(lambda(x,u+x=v)).
14 [] a<=0.
16 [14,demod,13] exists(lambda(x,a+x=0)).
17 [binary,16.1,9.1,demod,beta] a+e(lambda(x,a+x=0))=0.
18 [binary,17.1,6.3,demod,beta,11,beta,12,unit_del,1,3] $F.
```

A more difficult theorem involving order is

$$a \leq s(0) \rightarrow a = 0 \vee a = s(0).$$

This is an important principle about the ordering on the integers, often used in the proofs of more complicated inequalities. The proof (from nothing but induction, the definitions of addition and successor, and the definition of $\leq$) can be found on the Otter-λ website [7]

Our next example shows how Otter-λ can work with inequalities and a function defined by recursion. Otter-λ can prove

$$s(0) < a \rightarrow n < a^n.$$

The axioms used in the proof are induction, the definition of multiplication, the recursion equations for exponentiation, plus several other lemmas (lines 13,19–21,22, and 28). These lemmas were chosen by seeing what is needed in a

hand-constructed proof. Otter-λ is not able to prove this theorem from Peano's axioms and the recursion equations for exponentiation alone. Of course, we do expect that at some point this will happen: we don't expect to prove all theorems directly from Peano's axioms, we expect to use lemmas.

```
2 [] x*0=0.
5 [] x^s(y)=x*x^y.
9 [] -ap(y,0)|ap(y,g(y))|ap(y,z).
10 [] -ap(y,0)| -ap(y,s(g(y)))|ap(y,z).
13 [] -(u<v)|x*u<x*v| -(0<x).
15 [] 0<s(0).
17 [] 0<a.
19 [] x<y|y<=x.
20 [] -(y<=x)| -(x<y).
21 [] -(u<v)| -(v<=w)|u<w.
22 [] -(s(0)<z)| -(0<y)|s(y)<=z*y.
24 [] -(x<=0)|x=0.
25 [] s(0)<a.
26 [] x*s(y)=x*y+x.
28 [] 0+x=x.
29 [] x^0=s(0).
30 [] -(n<a^n).
```

Here is the rest of the proof, with the inductive Skolem term `g(lambda(x,x<a^x))` replaced by a constant `c`:

```
32 [binary,30.1,10.3,demod,beta,29,beta,5,unit_del,15] -(s(c)<a*a^c).
33 [binary,30.1,9.3,demod,beta,29,beta,unit_del,15] c<a^c.
35 [hyper,33,13,17] a*c<a*a^c.
37 [binary,32.1,19.1] a*a^c<=s(c).
41 [hyper,37,21,35] a*c<s(c).
42 [binary,41.1,20.2] -(s(c)<=a*c).
45 [binary,42.1,22.3,unit_del,25] -(0<c).
46 [binary,45.1,19.1] c<=0.
52 [binary,46.1,24.1] c=0.
67 [para_from,52.1.1,32.1.2.2.2,demod,29,26,2,28] -(s(c)<a).
76 [para_from,52.1.2,25.1.1.1] s(c)<a.
77 [binary,76.1,67.1] $F.
```

Inductive proofs involving non-integer variables

Real mathematics often involves proving theorems that contain some variables ranging over real numbers, or over members of an algebraic structure such as a group or ring, and other variables ranging over integers. The fact that Otter-*lambda* is not based on a fixed type system helps it to be able to deal with such

problems. On the other hand, since lambda logic and Otter-λ are untyped, the question arises as to whether the resulting proofs (if any) can be mapped back into a typed logic. That is, are we guaranteed that the proofs Otter-λ finds are correctly typeable?

The method used to answer this question is called *implicit typing*. It works like this: Assume that all the predicate and function symbols (including the constants) in a given input file can be given type specifications, by specifying a type for each argument position and a "value type" for the value of constants and functions. Predicates have value *boolean*. The variables are not typed. Each predicate and function symbol must have a *unique* type, except *Ap* and *lambda*, and they can have only two types: one for use when *Ap* is applied to propositional functions to produce propositions, and one for when *Ap* is used to apply to objects of some "ground type" and produce other objects of ground type. Under those assumptions, the proofs produced by Otter-λ will also be correctly typable. Detailed formulations and proofs of metatheorems with this import are in [5].

No nilpotents in an integral domain

An *integral domain* is a ring R in which $xy = 0$ implies $x = 0$ or $y = 0$. A *nilpotent* is a nonzero element of R such that $x^n = 0$ for some n, where x^n is defined by $x^o = 1$ and $x^{n+1} = x * x^n$, where o is the zero of the natural numbers and 1 is the unit element of R. The theorem in question here is that there are no nilpotents in an integral domain. Here is Otter-λ's proof of this theorem:

```
27 [] 1!=0.
28 [] x*y!=0|x=0|y=0.
30,29 [] pow(s(x),y)=y*pow(x,y).
32,31 [] pow(o,x)=1.
35 [] ap(x,o)| -ap(x,g(y,x))| -ap(x,y).
36 [] ap(x,o)|ap(x,s(g(y,x)))| -ap(x,y).
37 [] b!=0.
39 [] pow(n,b)=0.
41 [binary,39.1,36.3,demod,beta,32,beta,30,unit_del,27]
   b*pow(g(n,lambda(x,pow(x,b)=0)),b)=0.
43 [binary,39.1,35.3,demod,beta,32,beta,unit_del,27]
   pow(g(n,lambda(x,pow(x,b)=0)),b)!=0.
136 [binary,41.1,28.1,unit_del,37,43] $F.
```

This is not alleged to be a particular interesting proof for its own sake, though it is satisfying that Otter-λ proves it so cleanly. The reason for including this example is that it involves two types (or "sorts"): the type of natural numbers and the type of ring elements. In addition, the induction axiom involves the type of propositional functions on integers. It therefore illustrates the situation addressed by the implicit typing theorem. Specifically: how can we be assured that an Otter-λ proof of the no-nilpotents theorem is actually correctly

typeable? There are three ways: (1) We could examine the proof once it is in hand (either by hand or mechanically). (2) We could include `set(types)` in the input file, and replace one of the two constants o and 0 by a function term, for example $o = r(0)$ (since the metatheorem in [5] requires that all constants have the same type). (3) We could include `list(types)` in the input file, and specify `type(o,R)` and `type(0,N)`. Both (2) and (3) are done before we run Otter-λ and apply to any possible Otter-λ proof from that input file.

Proofs by induction involving simplification

We have also added to Otter the ability to make use of an external computation system; that of course is dangerous in general because of the possibility that the computation may depend on assumptions that are not valid at the point where the computation is applied. Otter-λ is linked to the computation modules of MathXpert [6], which do not suffer from this defect. This enables us to prove theorems by induction in which the induction step involves some computation. These steps appear in the Otter proof with the justification *Simplify*.

This feature is independent (with regard to the implementation) of lambda unification, in the sense that we could easily compile a version with only lambda unification, or only external computation linkage. Both are present in Otter-λ, but can be independently activated by switches in the input file: `set(lambda)` turns on lambda-unification and `set(simplify)` turns on external simplification.

Combining external simplification with lambda unification is interesting in the context of mathematical induction, since it enables Otter-λ to handle the examples of proof by induction that are usually given to students learning mathematical induction. We will show some examples of such proofs in this section.

Gauss's sum

This is the formula that Gauss supposedly rediscovered at age 12:

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}.$$

Here is the proof found by Otter-λ, after replacing the Skolem term

```
g(lambda(y,2*sum(z,z,0,y)=y* (y+1)))
```

by a constant `c` to improve the readability of the proof:

```
2 [] sum(u,v,x,x)=Ap(lambda(u,v),x).
4 [] sum(u,v,x,y+1)=sum(u,v,x,y)+Ap(lambda(u,v),1+y).
7 [] x+1!=y+1|x=y.
8 [] -Ap(y,0)|Ap(y,g(y))|Ap(y,z).
9 [] -Ap(y,0)| -Ap(y,g(y)+1)|Ap(y,z).
```

```
10 [] 2*sum(x,x,0,n)!=n* (n+1).
11 [binary,10,9,demod,beta,2,beta,beta,4,beta,simplify]
         2* (sum(x,x,0,c)+1+c)!= (1+c)* (2+c).
12 [binary,10,8,demod,beta,2,beta,beta,simplify]
         2*sum(x,x,0,c)= (1+c)*c.
22 [binary,11,7,simplify]
         2*c+2*sum(x,x,0,c)!=3*c+c^2.
29 [para_into,22,12,simplify] $F.
```

The first five formulas are axioms. Formula (10) is the negation of the goal. Formula (12) is recognizable as the induction hypothesis and (11) as the negation of what has to be proved in the induction step; note that the sum up to $c+1$ has already been split by *simplify* into the sum up to c and the $c+1$-st term. Now, (11) needs to be further simplified by multiplying out the right-hand side. But *simplify* hasn't already done that to (11) since it will only multiply out a product of sums if it occurs as a summand of another sum. So instead, (11) is resolved with (7), which in effect adds 1 to both sides of (11). Now on the right we have $(1+c)*(2+c)+1$, so *simplify* will multiply out and collect terms. Then there is a constant term 3 on both sides, which *simplify* subtracts from both sides. The result is formula (22). But in this formula, the induction hypothesis (12) occurs as a subformula, so it can be used (by paramodulation). The resulting left-hand side then simplifies to be identical to the right-hand side, so it resolves with $x = x$ to produce a contradiction.

Otter-λ has no trouble proving the formula

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}.$$

in a similar fashion.

Bernoulli's inequality

This is the inequality

$$1 + na \leq (1+a)^n \qquad \text{if } -1 < a.$$

Otter-λ successfully proves this by induction on n. This example is of particular interest for several reasons:

(i) It involves two types: reals and natural numbers. In order to formulate this theorem in such a way that the axioms can be correctly typed, so that the soundness theorems of [5] will apply, we must use a function symbol i for an injection $i : N \to R$. We use o for the zero in N and $0 = i(o)$ for the zero in R.[6]

[6]Technically, if we want the soundness theorem to apply *a priori* to any proof that Otter-λ might find, we should not use 0 at all, but only $i(o)$; and indeed if we make that replacement, the proof will still be found; but we can also simply observe that if we do use 0, the proof that is found is in fact correctly typed.

(ii) It offers an interesting interplay between algebra performed by MathXpert (external simplification) and algebra performed in the clausal search system, since at some point a law of exponents has to be used in the opposite direction from which simplification uses it. This is performed by demodulation.

(iii) Simplification is performed not only on terms of type N or R, but also on terms of type "proposition", specifically inequalities.

Here is the proof, with the Skolem term `g(lambda(x,a*i(x)+1<= (a+1)^x))` replaced by a constant `c` for readability.

```
2 [] x<=x.
3 [] x+1!=0.
7 [] -ap(y,o)|ap(y,g(y))|ap(y,z).
8 [] -ap(y,o)| -ap(y,s(g(y)))|ap(y,z).
10 [] -(x<y)| -(z<=x)|z<y.
12 [] -(x<=y)| -(y<x).
13 [] 0<a+1.
17 [] -(x<=y)| -(0<z)|z*x<=y*z.
19 [] -(x<y)|x+ -y<0.
22 [] x^o=1|x=0.
23 [] 0<=i(x).
25 [] i(o)=0.
26 [] i(s(x))=i(x)+1.
27 [] x^s(y)=x*x^y.
28 [] x*0=0.
30 [] 0+x=x.
33 [] x^ (y+z)=x^y*x^z.
34 [] x^1=x.
35 [] -(1+i(n)*a<= (1+a)^n).
36 [simplify,35] (a+1)^n<a*i(n)+1.
41 [binary,36,12] -(a*i(n)+1<= (a+1)^n).
43 [binary,41,8,demod,beta,25,28,30,beta,26,27]
     -(1<= (a+1)^o)| -(a* (i(c)+1)+1<= (a+1)* (a+1)^c).
44 [binary,41,7,demod,beta,25,28,30,beta]
     -(1<= (a+1)^o)|a*i(c)+1<= (a+1)^c.
85 [para_into,43,22,unit_del,2,3]
     -(a* (i(c)+1)+1<= (a+1)* (a+1)^c).
117 [para_into,44,22,unit_del,2,3]
     a*i(c)+1<= (a+1)^c.
119 [hyper,117,17,13]
     (a+1)* (a*i(c)+1)<= (a+1)^c* (a+1).
160 [para_into,43,22,unit_del,2,3,simplify,85,demod,33,34]
    (a+1)^c* (a+1)<a*i(c)+a+1.
298 [hyper,119,10,160]
    (a+1)* (a*i(c)+1)<a*i(c)+a+1.
299 [binary,298,19]
    (a+1)* (a*i(c)+1)+ -(a*i(c)+a+1)<0.
```

```
398 [binary,298,19,simplify,299]
     and(a!=0,i(c)<0).
404 [split and,398]
     i(c)<0.
412 [binary,404,12]
     -(0<=i(c)).
413 [binary,412,23] $F.
```

Let us go over this proof. First (at 36), the negated goal is simplified to an inequality. The next two steps, 43 and 44, represent the "decision" to attempt to prove this inequality by induction. The next two steps take care of the base case, so that 85 and 117 are recognizable as the induction step and the induction hypothesis, respectively. Hypothesis 17 says that we can multiply both sides of an inequality by the same positive quantity; line 119 is deduced by applying that, using the positive quantity $a + 1$. Line 160 is deduced from line 43 much as 117 was, but going further: the law 33, that is, $x^{y+z} = x^y * x^z$, is applied. Simplification uses this law in the right-to-left direction, so 160 would never be deduced by simplification alone; here it is deduced by using 33 as a demodulator. This is an important interplay between simplification (which has a tendency to be unidirectional, like demodulation) and the clausal-search mechanism, which can accumulate various equivalent forms of an expression. Once 160 is derived, trichotomy (10) and 119 give us 298, and subtracting the same term from both sides we get 299, which is

$$(a + 1) * (a * i(c) + 1) - (a * i(c) + a + 1) < 0.$$

This simplifies to a conjunction: $a \neq 0$ and $i(c) < 0$. Incidentally, I think one might wait a long time for this conclusion to come out without external simplification, using only paramodulation and demodulation! This conjunction is at first represented at the object level in Otter-λ as a term with functor `and` (line 398). But Otter-λ has an inference rule called `split and`, which converts a clause `and(P,Q) | R` to two clauses `P | R` and `Q | R`. This rule gives special meaning to the functor `and`. It is necessary because the mechanism for calling external simplification returns a single clause, not several clauses. After the use of "split and", the proof completes immediately, since the hypothesis that i maps integers to nonnegative reals has been contradicted.

Proving first-order formulas by induction

In order to use the clausal form of induction, the theorem to be proved must be a literal. However, in mathematical practice, we often need to prove a theorem by induction that is not directly expressed in that form. Here we take a well-known example (discussed in [9]): *Every natural number is either even or odd.* Since lambda logic is capable of defining the logical operators, we can represent *and* and *or* as constants. The meaning of these constants cannot be defined by

axioms in an input file, since these "axioms" would have to have variables in the place of a literal:

```
-or(x,y) | x | y.
-x | or(x,y).
-y | or(x,y).
```

This is not legal, either in Otter-λ or in lambda logic. Instead, to define disjunction and conjunction we have special inference rules, for example: "from $or(a,b) \mid c$ infer $a \mid b \mid c$." Four such rules for *and* and *or* are built into Otter-λ.[7] The following is Otter-λ's proof of the theorem that every number is even or odd. The predicates *even* and *odd* are defined in lines 1,2, and 10; the induction schema is given in lines 6 and 7, and the negated goal in line 13. For readability, the Skolem term $g(lambda(x, or(even(x), odd(x))))$ has been replaced by a constant c.

```
1 [] even(s(x))=odd(x).
2 [] odd(s(x))=even(x).
6 [] -Ap(y,0)|Ap(y,g(y))|Ap(y,z).
7 [] -Ap(y,0)| -Ap(y,s(g(y)))|Ap(y,z).
10 [] even(0).
13 [] -or(even(n),odd(n)).
16 [binary,13.1,7.3,demod,beta,beta,1,2]
   -or(even(0),odd(0))| -or(odd(c),even(c)).
17 [binary,13.1,6.3,demod,beta,beta]
    -or(even(0),odd(0))|or(even(c),odd(c)).
22 [split -or,16,unit_del,10] -or(odd(c),even(c)).
23 [split -or,22] -odd(c).
24 [split -or,22] -even(c).
27 [split or,17,unit_del,24,23] -or(even(0),odd(0)).
29 [split -or,27] -even(0).
30 [binary,29.1,10.1] $F.
```

Another approach to this example, more in the spirit of first-order logic, would be to introduce a predicate $R(x)$ for being even or odd. If we replace the use of *or* by axioms for R, then the formalization of first-order logic at the clause level can be avoided. This proof is as follows, again with a Skolem term replaced by a constant:

[7] It is not necessary to supply built-in rules for the quantifiers, as these can be defined by axioms in an input file when needed. This was illustrated above in the examples about order in Peano Arithmetic. Lambda logic with the new propositional constants and rules is a conservative extension of lambda logic, so by adding these rules we have not gone beyond the theoretical basis of Otter-λ in lambda logic.

```
1 [] even(s(x))=odd(x).
2 [] odd(s(x))=even(x).
6 [] -Ap(y,0)|Ap(y,g(y))|Ap(y,z).
7 [] -Ap(y,0)| -Ap(y,s(g(y)))|Ap(y,z).
10 [] even(0).
12 [] -R(x)|even(x)|odd(x).
13 [] -even(x)|R(x).
14 [] -odd(x)|R(x).
15 [] -R(n).
18 [binary,15.1,7.3,demod,beta,beta] -R(0)| -R(s(c)).
19 [binary,15.1,6.3,demod,beta,beta] -R(0)|R(c).
26 [binary,19.1,13.2,unit_del,10] R(c).
28 [binary,26.1,12.1] even(c)|odd(c).
31 [binary,18.1,13.2,unit_del,10] -R(s(c)).
36 [binary,31.1,14.2,demod,2] -even(c).
37 [binary,31.1,13.2,demod,1] -odd(c).
40 [binary,36.1,28.1] odd(c).
41 [binary,40.1,37.1] $F.
```

In general, both approaches can be used when formalizing a theorem in lambda logic. In other words, one can use the clause language as usual in first-order logic, or one can embed first order (and various higher-order) logics in the object level.

In order to give a good example of proving a quantified formula by induction, consider the problem of proving the principle of course-of-values induction from ordinary induction. Course of values induction can be expressed as an axiom schema in Peano Arithmetic as follows:

$$\forall z(\forall x < z(A(x) \rightarrow A(z)) \rightarrow \forall y A(y).$$

Replacing $A(x)$ by $Ap(a, x)$, and replacing $\forall x A(x)$ by $all(lambda(x, Ap(a, x)))$, we obtain a formalization in lambda logic. To prove course of values induction from ordinary induction, we take the following negated goal:

```
-implies(all(lambda(z,all(lambda(x,
 implies(and(x<z,Ap(a,x)),Ap(a,z)))))), Ap(a,c)).
```

Note that if we instead take a clausified form of this goal, there will be no hope of finding the "right" instance of induction by lambda unification, since there are no rules of inference that permit "reflecting" the clause level into the object level. But with this form, as usual lambda unification will find the right induction predicate, taking c for the masking term.[8] Once the correct instance

[8]If we had left $Ap(a, c)$ in the form $all(lambda(w, Ap(a, w))$ instead of using the Skolem constant c, then after a few steps essentially the same form will be reached, with a Skolem term $e(lambda(w, not(Ap(a, w))))$ instead of c.

of induction is found, it only remains to unwind the quantifiers and first-order connectives, using their definitions. This is just first-order reasoning (modulo alpha-equivalence)—no more lambda unification is required.

The theory of lists

We now take up list induction, to show that Otter-*lambda*'s inductive capabilities are not limited to number theory. Conveniently, Otter already supports list notation, with `[]` for the empty list and `[a|b]` for the list-making operation, usually read *a cons b*. One form of list induction is

$$P([]) \wedge \forall x, z(P(x) \rightarrow P([z|x]) \rightarrow \forall w P(w).$$

To express this in lambda logic, we replace $P(x)$ by $Ap(y, x)$. That yields

$$Ap(y, []) \wedge \forall x, z(Ap(y, x) \rightarrow Ap(y, [z|x]) \rightarrow \forall w Ap(y, w).$$

We bring this to the following clausal form:

```
-Ap(y,[]) | Ap(y,g(y)) | Ap(y,w).
-Ap(y,[]) | -Ap(y,[h(y)|g(y)]) | Ap(y,w).
```

Two Skolem functions h and g are required, instead of just one as in numerical induction. The recursive definition of `append` is

```
append([x|y],z) = [x|append(y,z)].
append([],z) = z.
```

These equations should be used as demodulators. We can now try to prove the associativity of `append` using the negated goal

```
append(n,append(b,c)) != append(append(n,b),c).
```

Again, the use of a constant named `n`, together with the command `set(induction)`, give Otter-λ a hint to choose `n` as the induction variable; this technique was introduced before backtracking over possible choices of the induction variable was implemented. The proof is found instantly; the presence of the lambda term

$$lambda(x, append(x, append(b, c)) = append(append(x, b), c)))$$

shows that the correct instance of induction was found. Here is the proof, after replacing Skolem terms with constants as follows:

$h(lambda(x, append(x, append(b, c)) = append(append(x, b), c)))$ becomes p
$g(lambda(x, append(x, append(b, c)) = append(append(x, b), c)))$ becomes q

```
1 [] append([],z)=z.
2 [] append([x|y],z)=[x|append(y,z)].
3 [] x=x.
6 [] -Ap(y,[])|Ap(y,g(y))|Ap(y,z).
7 [] -Ap(y,[])| -Ap(y,[h(y)|g(y)])|Ap(y,z).
8 [] append(n,append(b,c))!=append(append(n,b),c).
9 [binary,8.1,7.3,demod,beta,1,1,beta,2,2,2,unit_del,3]
   [p|append(q,append(b,c))]!=[p|append(append(q,b),c)].
10 [binary,8.1,6.3,demod,beta,1,1,beta,unit_del,3]
   append(q,append(b,c))=append(append(q,b),c).
13 [para_into,9.1.1.2,10.1.1]
   [p|append(append(q,b),c)]!=[p|append(append(q,b),c)].
14 [binary,13.1,3.1] $F.
```

Generalizing the theorem to be proved

We will examine the details of two problems from [9]. The first problem involves the function `rev` that reverses a list. It is defined by:

```
rev([]) = [].
rev(x|y) = append(rev(y),[x]).
```

The theorem to be proved is $rev(rev(x)) = x$. This example is given under the heading *Generalising a Sub-Term* on p. 873 of [9]. If we put in the additional hypothesis $rev(append(x, y)) = append(rev(y), rev(x))$, then Otter-$\lambda$ finds a proof instantly. The interesting question is whether Otter-λ can come up with this lemma by itself. And the answer is, it comes up by itself with the special case in which y is a one-element list:

$$rev(append(x, [z])) = append([z], rev(x).$$

This lemma is enough to finish the proof, and indeed, Otter-λ is able to prove $rev(rev(x)) = x$ unaided. Here is how it does it:

Specifically, the first attempt at induction derives the base case

$$rev(rev(c)) = c$$

and the negated induction step

$$rev(append(rev(c), [b])) \neq [b \quad | \quad c].$$

Now, paramodulating into the c on the right, from the induction hypothesis, Otter-λ gets

$$rev(append(rev(c), [b])) \neq [b \quad | \quad rev(rev(c))].$$

Then Otter-λ attempts to prove this by induction, and it selects as the masking term, $rev(c)$, generating the conjecture

$$rev(append(x,[b])) = [b \quad | \quad rev(x)].$$

This formula is readily proved by a straightforward induction. This works because Otter-λ's deterministic algorithm for selecting a masking subterm looks for a term (of weight 1 or 2) that occurs on both sides of the equation (when proving equalities by induction); and $rev(c)$ is such a term. This algorithm for selecting a masking subterm can be viewed as "generalization". If this seems *ad hoc*, bear in mind that backtracking selection of multiple unifiers with `max_unifiers` set to a small number would certainly retrieve this masking subterm. Here is Otter-λ's proof, with the complicated Skolem terms replaced by constants c, d, p, and q for readability. Notice the main induction hypothesis at line 14, the main induction step at line 13; the paramodulation at line 17 that enables the generalization to be found; the secondary (clever) induction is formulated at lines 22 and 23; the induction hypothesis can be used directly in the induction step, and only the definition of *append* is needed to finish off the proof. The associativity of *append*, which was originally in the input file, does not appear in the proof.

```
1 [] append([],z)=z.
2 [] append([x|y],z)=[x|append(y,z)].
3 [] rev([])=[].
4 [] rev([x|y])=append(rev(y),[x]).
7 [] x=x.
10 [] -Ap(y,[])|Ap(y,g(y))|Ap(y,z).
11 [] -Ap(y,[])| -Ap(y,[a(y)|g(y)])|Ap(y,z).
12 [] rev(rev(b))!=b.
13 [binary,12.1,11.3,demod,beta,3,3,beta,4,unit_del,7]
   rev(append(rev(d),[c]))!=[c|d].
14 [binary,12.1,10.3,demod,beta,3,3,beta,unit_del,7]
    rev(rev(d))=d.
17 [para_into,13.1.2.2,14.1.2]
    rev(append(rev(d),[c]))!=[c|rev(rev(d))].
22 [binary,17.1,11.3,demod,beta,1,4,3,1,3,beta,2,4,4,unit_del,7]
    append(rev(append(p,[c])),[q])!=[c|append(rev(p)),[q])].
23 [binary,17.1,10.3,demod,beta,1,4,3,1,3,beta,unit_del,7]
    rev(append(p,[c]))=[c|rev(p)].
27 [para_into,22.1.1.1,23.1.1,demod,2]
    [c|append(rev(p),[q])]!=[c|append(rev(p)),[q])].
28 [binary,27.1,7.1] $F.
```

Backtracking and non-determinism

All the example proofs in this paper have been obtained with a deterministic implementation of lambda unification. Recently, the ability to backtrack over

different lambda unifiers was added, so that the user can put the command `assign(max_unifiers,8)` in the input file, and a single lambda unification of $Ap(X, w)$ with t will return up to 8 unifiers (or whatever number is specified). Then we set out to test this improvement. We chose as our first example the theorem $x+(x+x) = (x+x)+x$ in Peano Arithmetic. This example was suggested by Bob Boyer, as one that would be too difficult for ACL2 (if associativity were not built into ACL2, as it is). The negated goal is $n + (n + n) \neq (n + n) + n$, and it seems that what one must do is select the two occurrences of n that are rightmost on the left and right, thus trying to prove $n + (n + z) = (n + n) + z$ by induction on z.

Surprisingly, Otter-λ proved this theorem *without* backtracking! How is that possible? Indeed Otter-λ first (fruitlessly) attempts to prove the theorem by induction on x. But eventually (clause 182 below), paramodulation and the Peano axioms for successor and the definition of addition generate the clause $n + (n + s(n)) \neq (n + n) + s(n)$. Once this clause appears, the deterministic implementation of lambda unification prefers the weight two term that appears on both sides: so it replaces $s(n)$ by a new variable and proves the theorem by induction on that variable. Here is the resulting proof, with the Skolem term $g(lambda(x, (n + n) + x = n + n + x))$ replaced by a constant c:

```
1 [] x+0=x.
3 [] x=x.
4 [] s(x)!=s(y)|x=y.
6 [] -ap(y,0)|ap(y,g(y))|ap(y,z).
7 [] -ap(y,0)| -ap(y,s(g(y)))|ap(y,z).
8 [] x+s(y)=s(x+y).
10 [] (n+n)+n!=n+n+n.
13 [binary,10.1,4.2] s((n+n)+n)!=s(n+n+n).
19 [para_into,13.1.1,8.1.2] (n+n)+s(n)!=s(n+n+n).
91 [para_into,19.1.2,8.1.2] (n+n)+s(n)!=n+s(n+n).
182 [para_into,91.1.2.2,8.1.2] (n+n)+s(n)!=n+n+s(n).
237 [binary,182.1,7.3,demod,beta,1,1,beta,unit_del,3]
    (n+n)+s(c)!=n+n+s(c).
238 [binary,182.1,6.3,demod,beta,1,1,beta,unit_del,3]
    (n+n)+c=n+n+c.
240 [para_from,238.1.1,8.1.2.1]
    (n+n)+s(c)=s(n+n+g(lambda(x,(n+n)+x=n+n+x))).
316 [para_into,237.1.2.2,8.1.1] (n+n)+s(c)!=n+s(n+c).
318 [para_into,240.1.2,8.1.2] (n+n)+s(c)=n+s(n+c).
319 [binary,318.1,316.1] $F.
```

On page 872 of [9] is another example, the special case of the associativity of `append` when all the variables are identified:

$$append(x, append(x, x)) = append(append(x, x), x)$$

Otter-λ cannot prove this theorem without backtracking for multiple masking subterms, even though, as we have seen above, it can easily prove the associativity of `append`; at least, not in fifteen minutes and 5000 generated clauses. Exactly why this is harder than the one-variable associativity of addition is not clear. However, if `max_unifiers` is set to 9, Otter-λ will backtrack through different choices of masking subterms in unification, generating the following possible choices of induction variable. The output was generated by a debugging trace, showing possible unifications of $Ap(X, z)$ with $append(a, append(a, a)) = append(append(a, a), a)$.

```
append(z,append(a,a))=append(append(a,a),a).
appead(a,append(z,a))=append(append(a,a),a).
append(z,append(z,a))=append(append(a,a),a).
append(a,append(a,z))=append(append(a,a),a).
append(z,append(a,z))=append(append(a,a),a).
append(a,append(z,z))=append(append(a,a),a).
append(z,append(z,z))=append(append(a,a),a).
append(a,append(a,a))=appead(appead(z,a),a).
append(z,append(a,a))=append(append(z,a),a).
```

The last result in this list is the one that makes the proof work. Here is the proof, with Skolem terms replaced by constants as follows:

$h(lambda(x, append(x, append(a, a)) = append(append(x, a), a)))$ becomes c
$g(lambda(x, append(x, append(a, a)) = append(append(x, a), a)))$ becomes d

```
1 [] append([],z)=z.
2 [] append([x|y],z)=[x|append(y,z)].
3 [] x=x.
7 [] -Ap(y,[])|Ap(y,g(y))|Ap(y,z).
8 [] -Ap(y,[])| -Ap(y,[h(y)|g(y)])|Ap(y,z).
9 [] append(a,append(a,a))!=append(append(a,a),a).
10 [binary,9.1,8.3,demod,beta,1,1,beta,2,2,2,unit_del,3]
   [c|append(d,append(a,a))]!=[c|append(append(d,a),a)].
11 [binary,9.1,7.3,demod,beta,1,1,beta,unit_del,3]
    append(d,append(a,a))=append(append(d,a),a).
18 [para_into,10.1.1.2,11.1.1]
   [c|append(append(d,a),a)]!=
   [c|append(append(d,a),a)].
19 [binary,18.1,3.1] $F.
```

Comparisons and Conclusions

Lambda logic and lambda unification have been implemented in the source code of Otter to produce Otter-λ. The theorems proved include some that are considered difficult for an inductive theorem prover. The conclusion to be drawn

from this work is not simply that Otter-λ is good at induction. The reason for its success is the underlying theory (lambda logic), the new unification algorithm (lambda unification), and the already existing strengths of the first-order prover Otter. Lambda logic and lambda unification are not extremely complex and can be used in other provers by anyone who wishes to implement them. Otter-λ serves to demonstrate the viability of this approach. Moreover, induction itself is only one example of an area in which reasoning about predicates and functions, in combination with strong first-order techniques, could be useful.

Therefore, the issue of an exact comparison between the performance of Otter-λ on induction problems and the performance of existing inductive provers is not vitally important for the evaluation and further application of this work. Before turning to that issue, I want to remark on the interplay in Otter-λ between brute force search and heuristics. It is important to notice that an implementation of the full non-deterministic lambda unification algorithm would generate as many unifiers as the induction hypothesis has subterms, and then perhaps try to prove many of those by induction again. Whether the proofs that are found with the present implementation would still be found, I do not know, but perhaps not, if the search space became filled with many useless attempted inductions. The present mix of heuristics and backtracking in Otter-λ seems to draw the line usefully between too many unifiers and too few. For example: the only reason (in induction problems) for choosing any masking term but a constant is to generalize the problem, as in the $rev(rev(x)) = x$ example. In this case, it seems unlikely that selecting a large masking term would be useful. The heuristic of selecting only a constant or term of weight 2 might be responsible for the fact that Otter-λ does not drown in a sea of hundreds of useless conclusions generated by multiple unifiers formed from arbitrary masking subterms. In the spirit of Otter, we have given the user as much control as possible over the proof search, by allowing a command of the form `assign(max_unifiers,9)`; also by using Otter's weight templates, the user can influence the process of generalizing an induction hypothesis, if desired.

In spite of the above disclaimer about the (in)significance of the question, it is still interesting to ask: Just how good is Otter-λ at induction, compared with existing inductive theorem-provers? Even without backtracking for multiple unifiers, Otter-λ is able to prove all the examples in Bundy's *Handbook* survey [9]. It may appear to some readers that the "Otter-λ proofs rely on numerous hints and have been 'coaxed out' of the system." This criticism may have been valid before the implementation of backtracking unification, when we did rely on heuristics in the prover to help it choose the right induction variable, e.g. telling it to prefer the letter n or m over a or b. But the current version of Otter-λ does not depend on such hints, and they may have been mostly unnecessary anyway, since upon removing them from input files, the proofs seem to still be found without backtracking.

Otter-λ can also prove at least some problems that are not provable with ACL2. We have mentioned $x + (x + x) = (x + x) + x$ (with addition renamed so that built-in associativy will not be available) and the corresponding example with *append* instead of addition. All this is accomplished without the use of

special heuristics such as those described in [9]; first-order search and paramodulation together with lambda unification form a powerful combination that needs no help. Bundy's "wave-front" heuristics are designed to control equations so that they are used in the direction required to convert the induction step to the induction hypothesis; his prover requires that equations be oriented.[9] In Otter-λ, paramodulation simply uses the equations in both directions; the extra conclusions do no harm, so there is no need to work hard to prevent them from being generated. ACL2 does not search for proofs, but constructs them by reducing goals, so it has to rely on heuristics for selecting an induction variable and for manipulating the induction goal by applying equations in the right direction. TPS [1], which is sometimes mentioned as an inductive prover, uses higher-order unification and generates many unifiers, yet it cannot automatically prove any theorem that requires a nested induction, i.e. one whose base case or induction step needs another induction.

However, the problems in the *Handbook* article form a small set of simple problems, and they are insufficient to decide the issue. Perhaps those other provers are more robust than Otter-λ, in that (i) they may be better at picking the correct induction principle automatically, including the correct induction variable or term, taking into account any existing definitions of recursive functions; or (ii) they may continue to work even if axioms are changed (e.g. so that addition is recursive in its first argument rather than its second); or (iii) they may perform better at industrial scale; or (iv) they may be better at theorems that have a lot of variables and require many inductions, such as $a + b + c + d = c + b + a + d$; or (v) they may be better on theorems that need generalization but don't mention the function needed in the generalization.

We address these possibilities one at a time. (i) Backtracking over multiple unifiers now allows Otter-λ to try every possible instance of induction formed by replacing (some set of occurrences of a) constant or term of weight 2 by an induction variable. (ii) We tried proving the associativity of addition with addition defined recursively on the first variable instead of the second (and all hints in the form of variable names removed). Otter-λ does that almost immediately, with very little searching required (see file PA-assocplus3.in at the website)–it does not even need multiple unifiers, in spite of the fact that its first attempt would be to use induction on the rightmost variable. The short and interesting proof is probably not one that a human would think of. It also proves the commutativity of addition easily when addition is defined by left recursion instead of right recursion, again without needing multiple unifiers. (iii) Otter-λ has not been tested at "industrial scale". It is intended for research. (iv) Otter-λ fails to prove that example; it generates half a million clauses and runs out of memory. Of course, it can be proved easily if commutativity and associativity are given, so the real problem is appropriate lemma generation, not the selection

[9] One should not conclude that because I have tried a different approach with Otter-λ, I have less admiration for Bundy's deep and beautiful work with these wave-front rules. In principle these rules could be used in combination with lambda unification and search–there is no reason why they cannot be combined in a single prover. For example, they could be used to guide the choice of the "right" lambda unifier.

of an appropriate instance of induction. (v) Otter-λ certainly cannot solve problems in that category, since its method for finding generalizations involves replacing subterms by variables.

Louise Dennis is currently developing a suite of test problems for inductive provers, with the aid of which, in the future, the performance of inductive theorem provers can be more quantitatively investigated.

References

[1] Peter B. Andrews, Matthew Bishop, Sunil Issar, Dan Nesmith, Frank Pfenning, Hongwei Xi, TPS: A Theorem Proving System for Classical Type Theory, *Journal of Automated Reasoning* **16**, 1996, 321–353.

[2] Aubin, R., Mechanizing Structural Induction Part I: Formal System. *Theor. Comput. Sci.* **9** 329–345 (1979).

[3] Aubin, R., Mechanizing Structural Induction Part II: Strategies. *Theor. Comput. Sci.* **9** 347–362 (1979).

[4] Beeson, M., Lambda Logic, in Basin, David; Rusinowitch, Michael (eds.) Automated Reasoning: Second International Joint Conference, IJCAR 2004, Cork, Ireland, July 4-8, 2004, Proceedings. Lecture Notes in Artificial Intelligence 3097, pp. 460–474, Springer (2004).

[5] Beeson, M., Implicit Typing in Lambda Logic, presented at the ESHOL workshop at LPAR-12, Dec. 2005.

[6] Beeson, M., MathXpert Calculus Assistant, software available from (and described at) www.HelpWithMath.com.

[7] Beeson, M. The Otter-λ website:

`http://www.MichaelBeeson.com/research/Otter-lambda/index.php`

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

[9] Bundy, Alan, The automation of proof by mathematical induction, Chapter 13 of [13].

[10] Bundy, Alan, *et. al.*, The Oyster-Clam system, in Stickel, M. E. (ed.) *10th International Conference on Automated Deduction* 647-648, Springer Lecture Notes in Artificial Intelligence **449** (1990).

[11] Kapur, D, and Zhang, H., An overview of Rewrite Rule Laboratory (RRL), *J. of Computer and Mathematics with Applications* **29** 2, 91–114, 1995.

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

[13] Robinson, Alan, and Voronkov, A. (eds.) *Handbook of Automated Reasoning, Volume II*, Elsevier Science B. V. Amsterdam, 2001. Co-published in the U. S. and Canada by MIT Press, Cambridge, MA.

[14] Wick, C., and McCune, W., Automated reasoning about elementary point-set topology, *J. Automated Reasoning* **5(2)** 239–255, 1989.

[15] Wos, Larry, and Pieper, Gail, *A Fascinating Country in the world of Computing*, World Scientific, Singapore (1999).