International Journal on Artificial Intelligence Tools
© World Scientific Publishing Company

OTTER-LAMBDA, A THEOREM-PROVER WITH UNTYPED LAMBDA-UNIFICATION

MICHAEL BEESON
Computer Science Department, San José State University
1 Washington Square
San José, California 95192, USA
beeson@cs.sjsu.edu

Support for lambda calculus and an algorithm for untyped lambda-unification has been implemented, starting from the source code for Otter. The result is a new theorem prover called Otter-λ. This is the first time that a resolution-based, clause-language prover (that accumulates deduced clauses and uses strategies to control the deduction and retention of clauses) has been combined with a lambda-unification algorithm to assist in the deductions. The resulting prover combines the advantages of the proof-search algorithm of Otter and the power of higher-order unification. We describe the untyped lambda unification algorithm used by Otter-λ and give several example theorems.

Keywords: Unification; lambda calculus; automated deduction.

Introduction

Our purpose in this paper is to demonstrate the successful combination of lambda unification with the well-known resolution-based theorem prover Otter [19]. What we call "untyped lambda unification" is similar to second-order or higher-order unification, but those algorithms are designed for use in typed theories, and we have chosen to use a new name for this algorithm, since it is a new algorithm and is applied in a different context.

The advantages of clause-based provers like Otter lie in the retention of deduced conclusions and the development of effective search strategies to generate and/or retain desired, rather than undesired, new conclusions. The advantages of type-based systems are the greater ease of describing mathematical concepts at a "higher level" and the availability of higher-order unification. By adding lambda unification to Otter, we have created a system offering a combination of both advantages.

The implementation of lambda calculus and an algorithm for lambda unification in Otter are described in the last sections of this paper. A soundness proof for the algorithm is given in[6]. That paper answers the question, *In what logic does Otter-λ find proofs?*. This paper reports on several examples intended to demonstrate some of the capabilities of Otter-λ, as we call our enhanced version of Otter (or when typography dictates, Otter-lambda). The main point is to explore the combination of lambda unification with a clause-based theorem prover using search strategies to generate new conclusions, as opposed to either a type-based prover (unaided by retention of many clauses, demodulation and paramodulation, and a variety of inference strategies) or a first-order, clause-based prover (unaided by higher-order unification and types).

First, we show that our enhancements give Otter-λ the ability to manipulate quantifiers at the clausal level, so that definitions involving quantifiers can be conveniently used in proofs. Second, we take up the algebraic part of Lagrange's theorem, to demonstrate how lambda unification enables the automatic construction of maps needed in algebraic proofs. Third, we give an example to show how Otter-λ can find a proof in algebra that uses mathematical induction, namely, there are no nilpotents in an integral domain. Fourth, we reproduce the well-known automatic deduction of Cantor's theorem, which appears to depend heavily on a typed logical system, to show that it can be done in Otter-λ; and that a similar approach in Otter-λ can yield Russell's paradox and fixed-point constructions, which cannot be formalized in type theory. Our fifth and last example is a proof by mathematical induction, directly from the axioms of Peano arithmetic, of the commutativity of addition. Both the basis case and the induction step of the main induction must in turn be proved by mathematical induction. Otter-λ is able to find all three instances of induction on its own, and complete the proof.

Input files and the proofs produced by Otter-λ for all these examples can be found on the Otter-λ web site. [7][a]

Lambda unification

We focus on the attempt to extend unification to instantiate variables for functions by means of λ-terms. All past research and implementation in this subject has involved typed formalisms. In that context, it is known as *higher-order unification.* What is novel in this paper is the implementation of (a version of) higher-order unification in an untyped formalism. We call our unification "untyped lambda-unification". In[23,24], Pietrzykowski gave an algorithm for higher-order unification, which we here call typed λ-unification. Huet[17] did not invent this algorithm, but his cited paper contains another important contribution and has become the classic reference. An encyclopedic treatment of type theory and second-order unification can be found in[1,14]; we shall say no more about the thirty years of research covered there. We note the following points about our implementation:

[a]To clear up some misconceptions about Otter-λ: It is a theorem-prover, built on Otter, but it is not a "library" that can be plugged into other provers, not even Otter, since it was necessary to modify Otter at several points to connect the new capabilities. On the other hand, it performs on first-order problems exactly like Otter–the new code is never invoked unless appropriate flags are set in the input file. Thus Otter files run in Otter-λ, and at the same speed, though they may use a few percent more memory since one data structure was slightly expanded.

- We do not implement the full (typed) higher-order λ-unification algorithm.
- Our algorithm can solve problems which are not solvable in a typed setting, e.g. finding a term X such that $X(y) = f(X(y))$.

Otter-λ recognizes the reserved words `lambda` and `Ap` (or synonymously `ap`). 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. To understand this paper, the reader will need to read both the examples and the algorithm for untyped lambda-unification. Either one is difficult to understand without the other, so it may be best to read them in parallel. Logically the algorithm definition should precede the examples, and the paper is organized that way, but it should also work to skip the rest of this section for now and return to it later.

A substitution σ is a λ-unifier of terms t and s if λ-logic [6] proves $t = s$. A λ-unification algorithm is an algorithm that finds one or more lambda unifiers, when given as input two terms t and s. We have (so far) implemented a single-valued version of second order unification. It therefore misses some unifiers that would be returned by a multiple-valued version, but it also solves some problems that cannot be solved in a typed theory, and it works efficiently, and it works in Otter. (We hope to extend the implementation in summer 2005 to return multiple unifiers.)

In this paper, as in Otter and Otter-λ, we distinguish between variables (which can be instantiated by unification) and constants (which cannot be instantiated) by a typographic convention: variable names begin with upper case or lower case x,y,z,u,v,w. Names beginning with other letters, such as f or c, are constants. The scope of a free variable is the clause in which it occurs. The scope of a λ-bound variable is the λ-term in which it occurs. The same variable will never occur both free and λ-bound in the same clause. (Nested λ-bindings of the same variable are allowed during unification, but for technical reasons are renamed when kept clauses are stored.) Technically, a *variable* refers to an integer *varnum* and a *context*, which is a table of pairs of the form (*term, context*) whose entries describe a substitution. Unification algorithms take as input two pairs (t, c) where t is a term and c is a context. In this section we refer to such a pair as a term, leaving the context implicit, or referring to it as *the context of* t. A term t depends on a variable y if t contains y or if t contains a variable that has been assigned, in the context of t, a value depending on y. When we say in the pseudocode below that $x := t$ we mean that an entry is made under the varnum of x in the context of x assigning x the value t. This setup is exactly as in first-order unification; see[19] for details.

The definition of λ-unification makes use of the concept of a variable y being *forbidden* to a variable x. This means that unification is not allowed to assign x a value depending on y. The context data structure is expanded to allow for storing, for each variable x, a list of variables forbidden to x. To forbid t to x means to add to the forbidden list of x, all free variables of t that have no values in the context of t, and all variables forbidden to those free variables of t that do have values assigned in the context of t.

Three functions are involved in our λ-unification algorithm: *unify2*, *unify_lambda*, and *subst*. The first two are are called from Otter's unification algorithm, *unify*; *subst* is mutually recursive with *unify*. When *unify* has to unify two terms s and t, if either one has functor Ap, then *unify2* is called, and if both have functor λ, then *unify_lambda* is called. We will give commented pseudocode for each of these three functions.

```
unify_lambda(lambda(x,t), lambda(y,s))
{ saveit = term currently assigned to x; // NULL unless x is a nested bound variable
  x:=y;        // assign y to x in the context of t
  forbid y to all free variables in t or s;
  unify(t,s);
  x:= saveit;
}
```

Example: $\lambda x.\, f(x)$ unifies with $\lambda y.\, f(y)$. No (net) variable assignment is made in either context.

The substitution algorithm *subst* is mutually recursive with unification. It takes three terms r, s, and t and returns a term (and a context). It is assumed r is a variable that does not occur in the contexts of s or t. The term is the result of substituting r for s in t. Here is pseudocode:

```
subst(r,s,t)
    { if(unify(r,s))
          { z = nextvar(t);       // get unused variable z in the context of t
            z:= r;                // assign z the value r in the context of t
            return t[s:=z];       // return t with s replaced by z
          }
      if(t is a variable and not λ-bound, and already assigned in the context of t to q)
          return subst(r,s,q);
      if(t is a variable or name)
          return t;
      n = arity of t;
      ans = new term with arity n and same symbol as t;
      for(i=0;i<n;i++)
          i-th argument of ans = subst(r,s,q);
      return ans;
    }
```

Example: substituting z for $g(X)$ in $g(Y)$, we return the substitution $X = Y$ and the term z. Below we write $t[s ::= r]$ for the term returned by *subst*.

We now give pseudocode for *unify2*. The code assumes that t and s are already dereferenced, i.e. are not variables that have values in their respective contexts.

```
int unify2(t,s)        // return value is 1 for success, 0 for failure
```

```
{ if(t has the form Ap(t1,t2) and s has the form Ap(s1,s2))
        if(unify(t1,s1) and unify(t2,s2))
              return 1;
  if(t has the form Ap(lambda(x,r),q))
        { z = a new variable;
          rename x in (a copy of) r to z;
          z:=q // in the context of t
          return unify(r,s); // perform a beta-reduction before unifying
          // if unification fails, destroy the copy of r before returning
          no memory used for this beta-reduction as context keeps track of it
        }
  if(s has the form Ap(lambda(x,r),q))
        return unify(s,t);
  if(t is a variable not occurring in s)
        { t:= s;
          return 1; // but don't fail if occurs check fails
        }
  if(s is a variable not occurring in t)
        { s:= t;
          return 1;
        }
  if(t has the form Ap(X,r))
        return unify2(s,t);
  if(s has the form Ap(X,r) and X is already assigned a value lambda(z,q))
        { // for simplicity omitting some details about clash of bound variables
          label z as "not lambda bound";
          z:= r;
          rval = unify(q,t);
          label z as "bound";
          return rval;
        }
  if(s does not have the form Ap(X,r) with X a variable)
        return 0;        // unification fails
  fix X and w with s = Ap(X,w);
  z = new variable not occurring in s or t;
  if(w is a variable and X does not occur in t)
        { b = getConstant(t);              // explained below
          rr = subst5(z,b,t);              // explained below
          X:= lambda(z,rr);
          return 1;        // success
        }
  else if(w is a variable and X does occur in t)
```

```
          { b = getMaskingSubterm(t);   // explained below
            rr = subst(z,b,t);
            X:= lambda(z,rr);
            return 1;        // success
          }
      else        if w is not a variable
          { rr = subst(z,w,t);
            if(X occurs in rr)
                 return 0;        // failure
            X:= lambda(z,rr);
            return 1;        // success
          }
    }
```

The above pseudocode outlines the algorithm, but leaves the functions *getConstant*, *getMaskingTerm*, and *subst5* still to explain. The two cases in question are unifying $Ap(X, w)$ with t, where w is a variable, and X either does occur in t (in which case *getMaskingTerm* is used) or does not occur in t (in which case *getConstant* is used). We illustrate with examples of each.

We first explain *getConstant*. Suppose t is $(a+b = b+a)$, as occurs when we try to prove the commutativity of addition by induction. Then *getConstant(t)* returns b, and *subst5* substitutes z for both occurrences of b (in this case), resulting in $X := \lambda z.\, a + z = z + a$, the right propositional function for this inductive proof. In general, however, there will be many choices of a possible constant, and then *subst5* has to choose which occurrences of the constant to substitute for. For example, if t is $(b + b) + b = b + (b + b)$ there are 65 possible choices of X. In the future, a more difficult implementation could allow *unify2* to return multiple unifiers, but we have had good success with the existing code, which just picks the rightmost constant and substitutes for all occurrences of it that do not occur inside Skolem terms from the induction axioms, if any. (The user indicates the Skolem terms to be avoided by using reserved symbols in the statement of the induction axioms.) This definition of subst5 allows Otter-λ to find the correct instances of induction for some nested inductive proofs, such as the commutativity of multiplication, where the basis case and induction step must themselves be proved by induction.

Now we describe *getMaskingSubterm*, which is used when unifying $Ap(X, w)$ with t where t does contain X. A *masking subterm* of t is a subterm r of t such that (i) r contains all the occurrences of X or variables forbidden to X in t, and (ii) r is not forbidden to w, and (iii) r does not contain any variables bound in t. We call such a term r a "masking subterm", because it masks the forbidden values of X. Any masking subterm could in principle be used to form a unifying substitution, as follows: We substitute the new variable z for r in t, obtaining R, and take $X := \lambda z.\, R$. Let σ be the substitution assigning this value to X and assigning

r to w. Then we verify that σ is a lambda unifier of $X(w)$ and t as follows:

$$\begin{aligned} Ap(X, w)\sigma &= Ap(X\sigma, w\sigma) && (1)\\ &= Ap(\lambda z.\, R), r) && (2)\\ &= R[z := r] && (3)\\ &= t && (4) \end{aligned}$$

The present implementation always chooses (if possible) a maximal masking subterm which respects any typing constraints the user has supplied in the input file under list(types); or if no list(types) is provided, a maximal masking subterm which is a second argument of an *Ap* term; that is, a masking subterm which occurs as a second argument of *Ap* and contains no masking subterm occurring as a second argument of *Ap*. In general we would get different potential unifiers for every masking subterm. There will sometimes be many of them. In the future the algorithm *getMaskingSubterm* can be altered or even made to return multiple values without affecting the soundness of the algorithm, as proved in[6]. In this way the tradeoff between efficiency and completeness can be further explored.

Finally, we give an example to illustrate the case where *subst* is called. To unify $Ap(X, s)$ with t, where s is a compound term: We create a new variable z and use *subst* to substitute z for s in t. We remind the reader of the convention that $t[x ::= z]$ (with a double colon) means the result of using *subst* to substitute z for x in t. If the result successfully eliminates the variables in t that are forbidden to X (that is, if $t[s ::= z]$ is not forbidden to X), then unification succeeds, giving X the value $\lambda z.\, t[s ::= z]$. *Example:* in attempting to unify $Ap(X, g(X))$ with $g(Y)$, we substitute z for $g(X)$ in $g(Y)$, so $t[x ::= z]$ is z and the substitution is $X = Y$, and the substitution returned by *unify2* is $X = \lambda z.\, z$.[b]

Regarding the implementation details: It was possible to make use of Otter's "Layer 1" tools, so that we had relatively little low-level implementation to do. Most of the code we wrote was directly related to the algorithm. We were able to allow Otter to manage memory, restoring trails after unification failures, etc. Otter uses the data structures described in[18] to perform unification without needing to allocate and deallocate memory for substitutions; we were also able to make good use of these data structures. The result was that we were able to add lambda-calculus, beta reduction, and second order (or "higher-order" if you like) unification to Otter by adding fewer than 2000 lines of C code. However, we did have to modify the indexing code that keeps track of possible subsuming terms, to take care of difficulties caused by λ-bound variables having scope less than the entire clause.

[b]In the example, we have missed the *other* solution, $X = \lambda z.\, g(Y)$. But we already knew that we are missing some of the unifiers produced by typed λ-unification. The question is, do we get the unifiers that we need to prove theorems of interest, and do we get them fast enough? Yes, we do, as shown by examples.

Example. We give an example in which untyped λ-unification succeeds, while typed λ-unification fails. The example is this:

$$Ap(X, y) = f(Ap(X, y))$$

See[14], p. 1035 for a discussion of why this fails, and how the failure is detected by one algorithm for typed λ-unification but not by another. Here we show how it works in untyped λ-unification. First we identify a masking subterm on the right: it is not $Ap(X, y)$ since this contains y, while a masking subterm on the right must be not be forbidden to y. For the same reason $f(Ap(X, y))$ is not a masking term. The only masking term is X itself. The unification algorithm therefore delivers the values

$$\begin{aligned} X &= \lambda z.\, f(Ap(z, z)) \\ y &= X \end{aligned}$$

Checking that this is a solution we find, using these equations,

$$\begin{aligned} Ap(X, y) &= Ap(\lambda z.\, f(Ap(z, z)), X)) \\ &= f(Ap(X, X)) \\ &= f(Ap(X, y)) \end{aligned}$$

Of course, this is the standard fixed-point construction in λ-calculus; the solution cannot be typed in simple type theory, and hence in a typed setting there is no solution to this unification problem. But in an untyped setting, untyped λ-unification can find fixed points.

Implementation of λ-calculus in Otter

Since second order unification involves lambda terms, it is first necessary to implement the lambda-calculus. It is the *untyped* lambda-calculus that is implemented. See[6] for the theoretical background and a soundness proof of the algorithm whose implementation is reported here. Otter-λ recognizes the reserved words `lambda` and `ap` (or synonymously `Ap`). 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] .
```

We give several more examples that demonstrate the correct handling of the λ calculus. For brevity we give only the negated goal from the input file and a brief comment. These examples have short proofs–a single unification leads to a unit conflict with the input $x = x$. The formulas shown are negated because one negates a goal when submitting it to Otter-λ. The symbol `!=` means "not equals".

`ap(lambda(x,lambda(y,x)),c) != lambda(z,c).`

alpha-conversion (renaming bound variables)

`lambda(x,a) != lambda(z,x).`

The first x is renamed and then unification succeeds with $x := a$.

```
ap(c,ap(lambda(x,ap(c,ap(x,x))), lambda(x,ap(c,ap(x,x))))) !=
ap(lambda(x,ap(c,ap(x,x))), lambda(x,ap(c,ap(x,x)))).
```

This verifies that $qq = \lambda x.\, f(xx)$ is a fixed point of f, i.e. $f(qq) = qq$. This example demonstrates that we are not working in a typed environment. Otter-λ is also capable of *finding* fixed points, not just verifying them as in this simple example.

Twenty-one years ago, in[18], the Argonne group published a description of the layered architecture which still underlies Otter. They remarked (p. 76), "The data structures of Layer 1 support greater generality (including, for example, the λ-calculus), but for the time being Layer 1 contains no manipulative procedures other than the basic ones." The "time being" lasted twenty-one years.

Quantification in a clausal theorem-prover

Although it is normal in resolution-based theorem provers to remove quantifiers by Skolemization before submitting a problem to the prover, this makes it difficult to deal with definitions that involve quantifiers, such as continuity of a function, or the "divides" relation on integers. In this section we show how this problem can be solved in Otter-λ. We show how to treat quantification 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 $\exists(\lambda n.P(n))$. Introducing a constant symbol $\exists$ this takes the explicit form $Ap(\exists, \lambda n.P(n))$. This definition of quantification allows one to work with quantified statements directly in Otter-λ. Since Otter-λ uses only characters on the keyboard, the constant is written `exists` in Otter-λ input files. One of the rules for $\exists$ can be expressed in an Otter-λ file (it is not hard-coded):

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

This works in Otter-λ as follows: if $Z(t)$ can be proved for any term t, then the literal $-Ap(Z,w)$ will be resolved away, using the substitution $w := t$. Then $\exists(\lambda x.Z(x))$ is deduced, which can be abbreviated to $\exists x.Z(x)$. No machinery is added to Otter-λ 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 $Ap(Z, x)$. Second-order unification will be called with x forbidden to Z, since the unification happens within the scope of the bound variable x. We get $Z := \lambda w.(2 * w = 6)$. 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,

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

That will be β-reduced to $2*w \neq 6$ so the clause finally generated will be $2*w \neq 6$. That resolves with $2 * 3 = 6$, producing a contradiction that completes the proof. [c]

If one wants to prove, say, the transitivity of the `divides` relation, then one needs the other law for the existential quantifier. Space does not permit the inclusion of this example, but it is posted on the Otter-λ website[7].

The algebraic part of Lagrange's theorem

In this section we use Otter-λ to find a proof of the algebraic part of Lagrange's theorem: the existence of the map $\lambda z.\, z * a$ from H to the coset Ha, where $a \in G$, and the inverse of this map.

To prepare this for Otter-λ, we assume that the range of the variables is the group G, so we do not need a unary predicate $G(x)$. (See the section on "implicit typing" for justification.) We use a unary predicate $H(x)$ for the subgroup H, and include the axioms that assert that the universe is a group under $*$ with identity

[c]We emphasize that this example is not meant to demonstrate prowess in number theory; we told the program that $2 * 3 = 6$ and concluded that 2 divides 6, which is trivial as number theory. Our point is rather to demonstrate how the program expanded a definition that involved a quantifier, and then used the second-order definition of quantifiers and second-order unification to automatically complete the proof. On the other hand, even though unmodified Otter incorporates a utility for Skolemizing first-order formulas, so that in some sense one can input quantifiers, it cannot do what is demonstrated here: use quantified formulas within the clause language while a resolution proof is being constructed.

e and inverse $i(x)$, and that H is closed under $*$ and inverse. We also include a constant a for a fixed element of G, and the definition of "there exists":

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

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-λ using a function symbol `f`, as follows:

```
f(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 `-f(t)` that arises. 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

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

Since we are using f for the characteristic function of Ha this becomes

$$\exists X \forall x (x \in H \to f(Ap(X, x)))$$

Introducing a Skolem function g, and replacing x by $g(X)$, the negated goal becomes the two clauses

$$\begin{aligned} H(g(X)). \\ -f(Ap(X, g(X))). \end{aligned}$$

The literal $-f(Ap(X, g(X)))$ demodulates and resolves with the existential axiom; the unification produces the substitution

$$Z := \lambda(x, and(H(x), x * a = Ap(X, g(X))))$$

and the resolution produces the unit clause

$$-Ap(\lambda(x, and(H(x), x * a = Ap(X, g(X)))), w).$$

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

$$-and(H(w), w * a = Ap(X, g(X))).$$

We now require a special rule of inference that connects the function symbol '*and*' to its logical meaning: from $-and(X, Y) \mid Z$ we can infer the clause $-X \mid -Y \mid Z$, where of course X or Y could be negative or positive literals, Z could be a list of literals, and X and Y do not necessarily come first in the clause. This rule of inference, along with a similar rule for the function symbol '*or*', has been implemented in Otter-λ. These rules are sound in the sense that they are provable in λ-logic ([6]). Thus Otter-λ obtains the clause $-H(w)|w * a \neq Ap(X, g(X))$. Next the literal $-H(w)$ resolves with $H(g(Z))$. The inferred clause is $g(Z) * a \neq Ap(X, g(X))$. The single literal in this clause can be unified with the equality axiom $x = x$. This results in unifying $g(Z) * a$ with $Ap(X, g(X))$. Since $g(X)$ is not atomic, the only way this unification can take place is if $g(Z) * a$ can be written as a term in $g(X)$. More precisely, substituting a new variable z for $g(X)$ in $g(Z) * a$ gives us $z * a$, unifying X with Z, so the top level call to unify returns the result $X = \lambda z.z * a$. That is exactly the desired result for the proof of Lagrange's theorem: $\lambda z.z * a$ is the desired map from H to the coset Ha.

The construction of the map $\lambda z.\, z * a$ required nothing but the definition of "there exists" and the existence of a binary operation. The next step is to prove the existence of an inverse map $Y = \lambda w.\, w * i(a)$, by giving Otter-$\lambda$ the negated goal

$$-f(Ap(X, g(X))) \mid -Ap(X, Ap(Y, r(Y))) = r(Y) \tag{5}$$

instead of only the first half, which was the goal in the example just completed. Here $r(Y)$ represents an arbitrary "constant" forbidden to Y, using the usual Skolemization technique. After finding $X = \lambda z.\, z * a$ as described above, the program deduces

$$Ap(\lambda z.\, z * a, Ap(Y, r(Y))) \neq r(Y)$$

which beta-reduces to

$$Ap(Y, r(Y)) * a \neq r(Y).$$

Now some group theory is finally needed. The program has to deduce

$$Ap(Y, r(Y)) \neq r(Y) * i(a).$$

using the axioms of group theory. Once that is done, this clause is resolved against $x = x$, which will cause the unification of $Ap(Y, r(Y))$ with $r(Y) * i(a)$. Since $r(Y)$ is forbidden to Y, the definition of unification causes a new variable w to be substituted for $r(Y)$ in $r(Y) * i(a))$, and the result is $Y = \lambda w.\, w * i(a)$ as desired. In order not to distract attention from the main issues of this paper, we put the group-theoretic lemma

$$u * x = v \mid u \neq v * i(x)$$

into the input file; but this is not necessary–one can comment that line out. Otter is generally good at this sort of thing, and unmodified Otter easily finds a four-line proof of this lemma.

This example, although it is only the algebraic part of Lagrange's theorem, still shows the usefulness of second-order unification operating as part of a resolution-based prover. The rest of Lagrange's theorem involves natural numbers, and the concepts of partition and counting. Otter-λ is also useful in those areas.

No nilpotents in an integral domain, solved in Otter

An *integral domain* is a ring satisfying $xy = 0 \to x = 0 \vee y = 0$. A *nilpotent* element is an x such that for some natural number n, $x^n = 0$. As a warming-up exercise in combining mathematical induction with algebra, we asked Otter-λ to prove that there are no nilpotents in an integral domain. It should discover the instance of induction required for itself. This experiment was successful, and there are several aspects of it worth discussing, before we move on to more difficult examples of mathematical induction.

First, the clausal formulation of induction. Experience has shown that this is neither familiar nor intuitive to many people, even experts in automated deduction, and therefore we spell it out. Second, the issue of "sorts" and its solution by "implicit typing". Although this also arises in connection with ordinary Otter (or other first-order provers), again many people are not familiar with the concept. Third, the use of λ-unification to instantiate the variable property in the induction axiom, thus "finding the correct instance of induction" to use. This is the important point of the example–the other two points are just prerequisites.

We take up the clausal form of induction now. We start with the usual formulation of induction, using a variable y for a predicate of natural numbers:

$$Ap(y, 0) \wedge \forall x(Ap(y, x) \to Ap(y, s(x))) \to Ap(y, z)$$

We now reduce it to clausal form. The universally quantified x in the antecedent has to get Skolemized as $g(z, y)$, where g is a new Skolem function symbol. As an intermediate step we obtain

$$Ap(y, 0) \wedge Ap(y, g(z, y)) \to ap(y, s(g(z, y))) \to Ap(y, z).$$

Changing the first implication $P \to Q$ to $Q| - P$ (where as usual in the clause language the vertical bar means "or" and the dash means "not"), we have

$$Ap(y, 0) \wedge (-Ap(y, g(z, y))|Ap(y, s(g(z, y)))) \to Ap(y, z)$$

and doing that again we get

$$-Ap(y, 0)|(Ap(y, g(z, y)) \wedge -ap(y, s(g(z, y))))|Ap(y, z)$$

Because of the conjunction, this splits into two clauses:

$$-Ap(y, 0)|Ap(y, g(z, y))|Ap(y, z).$$
$$-Ap(y, 0)| - Ap(y, s(g(z, y)))|Ap(y, z).$$

Note that if we humans choose a particular formula $P(z)$ to replace $Ap(y, z)$, we can express induction on one particular P in ordinary first order logic. People say that Otter cannot do induction, but actually it can do induction, if you tell it what instance of induction to use. In the case of the problem of "no nilpotents in an integral domain", obviously we should take $P(x, z)$ to be $x^z \neq 0$. If we do that, and give Otter the two resulting clauses to use and the negated goal $a^n = 0$, it is not difficult to set Otter's inference rules and parameters so that it finds a proof. However, one may object that with this formulation we have only proved the theorem for integral domains whose underlying set is the natural numbers; if we want something more general we ought to have used unary predicates for the ring and for the natural numbers. We will see in the next section that this is not so.

Implicit Typing

In the "no nilpotents in an integral domain" problem, there are three kinds, or "sorts", of things involved: ring elements, natural numbers, and predicates of natural numbers (because the induction axiom is stated with a variable for a predicate, and we expect Otter-λ to instantiate that variable). Should we use unary predicates to define these three "sorts"? Generally we wish to avoid using such unary predicates if possible. The method of "implicit typing" shows that under certain circumstances we can dispense with these unary predicates. One assigns a type to each predicate, function symbol, and constant symbol, telling what the sort of each argument is, and the sort of the value (in case of a function; predicates have Boolean value). Specifically each argument position of each function or predicate symbol is assigned a sort and the symbol is also assigned a "value type" or "return type". For example, in this problem the ring operations $+$ and $*$ have type $R \times R \to R$, which we might express as $type(R, +(R, R))$. If we use N for the sort of natural numbers then we need to use a different symbol for addition on natural numbers, say $type(N, plus(N, N))$, and we need to use a different symbol for 0 in the ring and *zero* in N. The symbol Ap in this problem satisfies $type(N, Ap(P, N))$, where P represents the sort of predicates. The Skolem symbol g in the induction axiom satisfies $type(N, g(N, P))$. We call a formula or term "correctly typed" if it is built up consistently with these type assignments. Note that variables are not typed; e.g. $x + y$ is correctly typed no matter what variables x and y are. But when a variable occurs in a formula, it inherits a type from the term in which it occurs, and if it occurs again in the same clause, it must have the same type at the other occcurence for the clause to be considered correctly typed. Once all the function symbols, constants, and predicate symbols have been assigned types, one can check (manually for now) whether the clauses supplied in an input file are correctly typed. Then one observes that if the rules of inference preserve the typing, and if the axioms are correctly typed, and the prover finds a proof, then every step of the proof can be correctly typed. That means that it could be converted into a proof that used unary predicates for the sorts. Hence, if it assists the proof-finding process to omit these unary predicates, it is all right to do so. This technique was introduced long ago in[28], but McCune says it was already folklore at that time. It implies that the proof Otter finds using an input file corresponding to the above formulation actually is a valid proof of the theorem, rather than just of the special case where the ring elements are the natural numbers.

No nilpotents in an integral domain, solved in Otter-λ

Using Otter-λ instead of Otter, we just give the prover the general formulation of induction, with a variable for the predicate. Recall those two clauses:

$$-Ap(y,0)|Ap(y,g(z,y))|Ap(y,z).$$
$$-Ap(y,0)|-Ap(y,s(g(z,y)))|Ap(y,z).$$

Otter-λ will apply binary resolution to the negated goal $a^n = 0$ and the third literal in each of the two induction clauses. Untyped λ-unification will get the inputs $Ap(y,z)$ and $a^n \neq 0$ as terms to be unified. The result will be to instantiate y to $\lambda z.\, a^z \neq 0$. The term $-Ap(y,0)$ will become $-Ap(\lambda z.\, a^z \neq 0), 0)$, which will beta-reduce to $a^0 \neq 0$, readily recognized as the basis case of the induction. This case can be solved (resolved away) using the axioms $x^0 = 1$ and $1 \neq 0$. There then remain two clauses with one literal each. Writing c to abbreviate $g(n, \lambda(z.\, a^z \neq 0))$, the remaining clauses can be written $a^c = 0$ and $a^{s(c)} \neq 0$. This is evidently the induction step, set up for proof by contradiction. The Skolem term c can be thought of as an arbitrary “constant”. Indeed Otter-λ finds a proof.

But does this proof actually prove the theorem? We ask whether implicit typing remains a valid technique in Otter-λ, where beta-reduction and lambda-unification are used in the inference process. A λ-term $\lambda x.\, t$ (which in Otter-λ is written `lambda(x,t)`) is correctly typed if t is correctly typed. In that case if x occurs in t it inherits a type from its parents in t (and since t is correctly typed it inherits the same type from all occurrences). If that type is R and the return type of t is S then $\lambda x.\, t$ has the return type $R \to S$ of functions from R to S. We can also ask whether the algorithm for untyped lambda-unification preserves correct typings. For now, suffice it to say that in the case of the “no nilpotents” problem, lambda-unification is used only to instantiate y to $\lambda z.\, a^z \neq 0$. (This is the 0 of the ring, not the *zero* of N, so that $Ap(y,z)$ is made equal to $a^z \neq 0$. In the induction clauses, Ap can be correctly typed by $type(Boolean, Ap(Pred, N))$, where $Pred$ is the type of predicates on N, i.e., $N \to Bool$. Thus y inherits the type $Pred$ from its occurrences in (each of) those clauses. The term $\lambda z.\, a^z \neq 0$ also has type $Pred$, so at least in this proof, implicit typing does apply.

This argument illustrates the point that while we can sometimes guarantee in advance of running a prover (Otter or Otter-λ) that a proof obtained from a particular input file will certainly be correctly typable[d] , it is also possible to run the prover first, and then inspect the resulting proof, verifying that it can be correctly typed. In the example at hand, it is possible to carry out an *a priori* argument, but we do not do that here and now. We simply want to illustrate how Otter-λ can find the correct instance of induction, and how implicit typing can be used. As of June

[d]There is an essay about implicit typing on the Otter-λ website[7] containing a theorem about this, but it is too long to include here.

2004, Otter-λ can make use of an optional list of typings of function symbols and predicates supplied in the input file to ensure that it produces only unifiers that respect those typings.

Cantor's theorem and Russell's paradox

One of the most celebrated examples of the use of higher-order unification in theorem-proving is the automatic generation of the "diagonal" proof of Cantor's theorem (see e.g.[4]). Since this theorem and its proof seem at first glance to be heavily dependent, on type theory, we want to show that Otter-λ also can find the proof. We use the method of "implicit typing" discussed above to formulate this problem. Intuitively, we have a set (or type) α and its power set $P(\alpha)$, and we suppose for proof by contradiction that we have a map c from α onto $P(\alpha)$. Then the statement to be proved contradictory is

$$\forall x \in P(\alpha) \exists j \in \alpha \forall z \in \alpha (Ap(x,z) = Ap(Ap(c,j),z)).$$

As usual we introduce a Skolem function $j = J(x)$. Then the "negated goal" becomes

$$Ap(x,z) = Ap(Ap(c,J(x)),z).$$

We also need the following two axioms:

$w \neq not(w)$	definition of negation
$u \neq v \vert v = u$	symmetry of equality

The (implicit) typings in this problem are as follows:

$\alpha\ J(P(\alpha))$	J maps $P(\alpha)$ into α
$P(\alpha)\ c(\alpha)$	c maps α into $P(\alpha)$
$Prop\ Ap(P(\alpha),\alpha)$	Ap is proposition-valued on $P(\alpha)$
$P(\alpha)\ Ap(\beta,\alpha)$	where β is the type of maps from α to $P(\alpha)$
$Prop\ not(Prop)$	the negation of a proposition is a proposition

These types are *implicit*, i.e. do not form part of the input to the prover. Note that Ap does not have a unique implicit type in this problem, unlike in the problem about nilpotents in integral domains. Nevertheless the second argument of Ap always has type α. We do not, then, know *a priori* that an Otter-λ proof of contradiction from the goal given above will be correctly typable, but this can easily be verified by inspection if it is true for a given proof. Otter-λ finds such a proof instantaneously: It resolves the equation $Ap(x,z) = Ap(Ap(c,J(x)),z)$ with $u \neq v|v = u$, producing the new conclusion

$$Ap(Ap(c,J(x)),y) = Ap(x,y).$$

This is then resolved with $w \neq not(w)$; the first step assigns w the value $Ap(Ap(c,J(x)),y)$, and then we have the unification problem

$$Ap(x,y) = not(Ap(Ap(c,J(x)),y)).$$

The "masking term" on the right, i.e. the smallest term that contains all occurrences of x, and occurring as a second argument of Ap, is $J(x)$. Therefore our unification algorithm selects $J(x)$ and y to be replaced by the new λ-bound variable, and takes $x = \lambda u.not(Ap(Ap(c,u)),u))$ and $y = J(x)$, or explicitly, $y = J(\lambda u.not(Ap(Ap(c,u)),u))$. We check this solution: $Ap(x,y)$ then beta-reduces to $not(Ap(Ap(c,y)),y))$, but $y = J(x)$, so we have $not(Ap(Ap(c,J(x)),y))$, which is the right-hand side of the unification problem. Yes, they are equal. The Otter-λ input file and resulting proof are available online at[7]. It is apparent that the Otter-λ proof is correctly typable.

It is quite amusing that this very same proof is also a proof of Russell's paradox. To see this, we use the usual representation of sets in λ-calculus, namely we regard $u \in v$ as an abbreviation for $Ap(v,u)$, where v is a $Prop$-valued mapping. Then if we add the axioms $Ap(c,x) = x$ and $Ap(J,x) = x$ to the input file, the term $x = \lambda u.not(Ap(Ap(c,u)),u))$ is just another notation for $\{u : u \notin Ap(c,u)\} = \{u : u \notin u\}$. That is just the Russell set R, automatically defined by the magic of higher-order unification. But note that the Russell paradox cannot even be formulated in type theory–that was, after all, the point of Russell's invention of type theory. Of course, we could remove c and J entirely, rather than adding extra axioms saying they are the identity, but leaving them in shows the relation between Cantor's theorem and Russell's paradox more clearly.

We can modify this input file in two different ways to escape Russell's paradox; these modifications correspond directly to the traditional "solutions" of the paradox. First, we can simply regard the paradox as showing that there is no fixed point of *not*: it is the axiom $w \neq not(w)$ that is suspect. We can say that this holds only for propositions. Alternately we can say that the problem is that Ap is not total. We need to allow for propositions that "never converge" to true or false. We can do that by introducing a "logic of partial terms", replacing the equality axiom $x = x$ by $x = x|E(x)$, where $E(x)$ means "x is defined" or "x exists". Then we get no contradiction unless we also assume $E(R)$; in other words, the "paradox" shows that the expression "defining" the Russell set is actually undefined; or one may choose to express this by saying the Russell set does not exist.

This latter fine distinction is reflected in different logics of partial terms. More than one such logic has been studied (see[5], p. 97.), and now we are able to deal with them in Otter-λ. Feferman's theories of classes and operations (see[5], Chapter X), which have been put forward as a competitor to type theory for the formalization of mathematics, are based on a logic of partial terms (or the equivalent). Now, higher-order unification can be used with these theories, not only with type theory. See[6] for more on this subject.

Commutativity of addition proved from Peano's axioms

We now give an example to illustrate search and untyped λ-unification working in concert. The example is, to prove the commutativity of addition from Peano's axioms. Of course, we don't need the axioms about multiplication–all we need is the definition of addition and the induction axioms. Specifically the axioms and the negated goal are shown in the first six lines of the proof:

```
1 [] x+0=x.
10 [] x=x.
13 [] -ap(y,0)|ap(y,g(z,y))|ap(y,z).
14 [] -ap(y,0)| -ap(y,s(g(z,y)))|ap(y,z).
15 [] x+s(y)=s(x+y).
16 [] a+n!=n+a.
```

At the next step, Otter-λ finds the correct instance for the main induction:

```
 a!=0+a|a+s(g(n,lambda(x,a+x=x+a)))!=s(g(n,lambda(x,a+x=x+a)))+a.
```

The occurrence of this lambda term shows that Otter-λ is attempting to prove $a + x = x + a$ by induction on x. Later in the proof Otter-λ constructs and uses the terms `lambda(y,y=0+y)`, indicating that it will try to prove the basis case of the main induction by induction on the other variable, and

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

Writing `c` to abbreviate the inner Skolem term, this is `lambda(y,s(c+y)=s(c)+y)`, showing that it has identified the crucial lemma: it will try to prove `s(c+y) = s(c)+y` by induction on `y`. When I proved this theorem by hand, I chose to prove `c+s(y) = s(c)+y` by induction on `y`, which is just one step away from Otter-λ's choice, by the definition of addition. Here are the remaining twenty steps of Otter-λ's proof of the theorem, edited for readability by substituting `c`, `b`, and `d` for Skolem terms involving the three instances of induction just mentioned. The unedited proof can be read (or regenerated) at[7]. The search was modest by Otter's standards: 8980 clauses were generated, and the proof took two seconds of CPU time to find. Even so, finding it unaided direct from Peano's axioms is an interesting achievement.

```
17 [binary,16.1,14.3,demod,beta,1,beta] a!=0+a|a+s(c)!=s(c)+a.
18 [binary,16.1,13.3,demod,beta,1,beta] a!=0+a|a+c=c+a.
35 [binary,17.1,14.3,demod,beta,1,beta,unit_del,10] a+s(c)!=s(c)+a|s(b)!=0+s(b).
36 [binary,17.1,13.3,demod,beta,1,beta,unit_del,10] a+s(c)!=s(c)+a|b=0+b.
56 [binary,18.1,14.3,demod,beta,1,beta,unit_del,10] a+c=c+a|s(b)!=0+s(b).
57 [binary,18.1,13.3,demod,beta,1,beta,unit_del,10] a+c=c+a|b=0+b.
85 [para_from,57.2.2,15.1.2.1] 0+s(b)=s(b)|a+c=c+a.
102 [para_from,36.2.2,15.1.2.1] 0+s(b)=s(b)|a+s(c)!=s(c)+a.
716 [para_from,85.1.1,56.2.2,unit_del,10,factor_simp] a+c=c+a.
719 [para_from,716.1.1,15.1.2.1] a+s(c)=s(c+a).
735 [para_from,719.1.1,15.1.1] s(c+a)=s(a+c).
759 [para_into,735.1.2,15.1.2] s(c+a)=a+s(c).
1598 [para_from,102.1.1,35.2.2,unit_del,10,factor_simp] a+s(c)!=s(c)+a.
```

```
1612 [para_into,1598.1.1,759.1.2] s(c+a)!=s(c)+a.
1620 [binary,1612.1,14.3,demod,beta,1,1,beta,unit_del,10] s(c+s(d))!=s(c)+s(d).
1621 [binary,1612.1,13.3,demod,beta,1,1,beta,unit_del,10] s(c+d)=s(c)+d.
1634 [para_into,1621.1.1,15.1.2] c+s(d)=s(c)+d.
1663 [para_from,1634.1.1,15.1.2.1] c+s(s(d))=s(s(c)+d).
1710 [para_into,1620.1.2,15.1.1] s(c+s(d))!=s(s(c)+d).
1764 [para_into,1663.1.1,15.1.1] s(c+s(d))=s(s(c)+d).
1765 [binary,1764.1,1710.1] $F.
```

Related work

For references to the hundreds of technical papers about higher-order unification in typed formalisms, see[1,14]. As far as I can determine, there has been no previous implementation of lambda-calculus and second-order unification in an untyped, resolution-based, first-order theorem prover. Therefore the related work consists mostly of implementations of higher-order logic. These include:

- PVS, developed at SRI under the direction of N. Shankar[26]
- HOL-Light, which was written by John Harrison[15,16], and is currently being used by him at Intel's Portland facility
- NuPrl, developed at Cornell under the direction of Constable[11]
- Isabelle, developed by Paulson and Nipkow[22]
- Coq, developed at INRIA, and also in use in Nijmegen[12]
- λ-Prolog[21]
- TPS[20,2,4]

PVS, HOL-Light, NuPrl, Isabelle, Theorema, and Coq are primarily proof-checkers, not proof-finders, although all of them have some ability to fill in the small steps of a proof automatically. Isabelle uses higher-order unification (see[22], p. 77). Coq is based on the Calculus of Constructions[13]. A discussion of the theoretical basis of Coq and a history of its development can be found in the introduction to[12], which states that "the salient feature which clearly distinguishes our proof assistant ... is its possibility [ability] to extract programs from the constructive content of proofs." Coq uses at least some higher-order unification[e], but has no significant proof-search capabilities. Theorema inherits lambda-calculus from Mathematica (*Function[x,t]* is Mathematica notation for $\lambda x.\, t$), but the deduction apparatus of Theorema is based on first-order natural deduction. The architecture of Theorema allows for multiple (special-purpose) provers, but no prover using higher-order unification has been designed or implemented. NuPrl does not use higher-order unification; its proof-

[e] I could not find this stated in the Coq tutorial or manual, but searching the Coq source code one finds several mentions of it, e.g. the comment in the implementation of the *apply* tactic: "Last chance: if the head is a variable, apply may try second order unification". On the other hand Coq often reports "Cannot solve a second-order unification problem," but one cannot be certain what that implies about the implementation details, if anything.

checking method is based on term reduction of a proof term constructed by the user-directed application of tactics. That same description applies to HOL-Light. PVS is similar, but with emphasis placed on the inclusion of decision procedures, and more recently, model-checking[26]. PVS has mainly been applied to the verification of programs, e.g. microcode for a commercial chip. λ-Prolog extends Prolog to a larger class of formulae using higher-order techniques, but like Prolog, it is based on a depth-first search algorithm rather than an architecture that maintains a database of generated conclusions and uses strategies to control the retention of conclusions and the selection of the parents for the next generation of conclusions. It does make use of higher-order unification. The table below summarizes this information. ACL2[8] and Theorema[10] are included, even though they do not implement higher-order logic.

System	**Higher order unification?**	**Accumulates Conclusions?**	**Proof Search?**	**Can vary rules and strategies?**
PVS	matching	n	some	y
HOL-Light	y	n	MESON_TAC	n
Isabelle	y	n	n	y
NuPrl	n	n	n	n
Coq	y	n	n	n
Theorema	n	n	some	y
λProlog	y	n	HH formulas only	n
TPS	y	y	y	some
ACL2	n	n	y	n
Otter	n	y	y	y
Otter-λ	some	y	y	y

Of the systems mentioned, the one most relevant to our work is TPS. In[3], the creators of TPS say,

> TPS combines ideas from two fields which, regrettably, have not achieved much cross-fertilization. On the one hand, there is the traditional work in first-order theorem proving using such methods as resolution, model elimination, or connection graphs. On the other hand we find avant-garde proof-checkers and theorem provers for type theories of a variety of flavors, mostly centered around interactive proof construction with the aid of tactics.

The same could be said about Otter-λ. The difference is that TPS imports inference mechanisms from first-order provers into a typed system, while we import unification methods from higher-order logic into a first-order system. Is there any point in taking this different approach to the problem? After all the higher-order methods are complete for first-order logic, so it might be (and has been!) alleged that it is a waste of energy to try to incorporate higher-order methods in an existing first-order prover like Otter.[f] Here is another quote from[3]:

> Naturally, TPS can be used to prove theorems of first-order logic, but we focus mainly on examples from higher-order logic. (Apart from the development of path-focused duplication, a relatively small part of the development effort for TPS has been devoted thus far to certain basic issues of search which are important for first-order logic.)

Specifically, those "issues of search" include having a variety of rules of inference useful for different problems, and having a variety of strategies available for controlling the search, such as pick-and-purge, weight templates, etc., as well as making use of the indexing and memory management techniques provided in the layered architecture on which Otter is based. It is, however, clear that the first-order capabilities of TPS, and even more of the other systems listed, do not at present compare to those of Otter, in part because they lack the features just mentioned. We do not mean to criticize TPS or any of the systems above; the facilities that TPS offers for higher-order unification are probably superior to what we have added to Otter. Decades of work by many very capable experts have gone into these systems and admirable results (commensurate with the efforts) have been obtained. We are only trying to make clear where Otter-λ fits in. In the conclusion of [4], the authors say

> There is much to be done in the development of methods for higher-order theorem proving .... Some major areas where work is needed are: the basic mechanisms of searching for matings; the efficiency of higher-order unification; the treatment of equality; the introduction of rewrite rules....

Three of these four are addressed in the present work by using the existing solution to these problems in Otter.

Regarding the specific example of Lagrange's theorem: A proof of Lagrange's theorem has been checked by the use of Nqthm[29]. Curiously, this proof is not the usual proof, but a proof by induction on the order of the subgroup. Lagrange's theorem has also been formalized (using set theory) in the Mizar library[27].

Acknowledgments

I am grateful to Bill McCune for prompt and lucid answers to questions about Otter, and to Larry Wos and Freek Wiedijk for discussions and suggestions. This research was supported by NSF grant number CCR-0204362.

References

1. Andrews, P., Classical type theory, Chapter 15 of[25].

[f] I have been told that the interest of the creators of TPS was more in closing the gap between automatic and interactive theorem proving, than in closing the gap between first-order and higher-order theorem proving.

2. Peter B. Andrews, Sunil Issar, Dan Nesmith, and Frank Pfenning, The TPS Theorem Proving System, in: Stickel, Mark (ed.), *10th International Conference on Automated Deduction*, 641–642, Lecture Notes in Artificial Intelligence **449**, Springer-Verlag, (1990).
3. Peter B. Andrews, Matthew Bishop, Sunil Issar, Dan Nesmith, Frank Pfenning, Hongwei Xi, TPS: A Theorem Proving System for Classical Type Theory, Carnegie Mellon University Department of Mathematics Research Report 94-166A, February, 1995. Available at http://gtps.math.cmu.edu/tps94-report-94-166a.pdf, while[4] is not available electronically.)
4. 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.
5. Beeson, M., *Foundations of Constructive Mathematics*, Springer-Verlag, Berlin/ Heidelberg/ New York (1985).
6. Beeson, M., Lambda Logic, accepted for publication in the proceedings of IJCAR 2004. Also available for download near the bottom of www.cs.sjsu.edu/faculty/beeson/Papers/pubs.html, along with a supplement containing two proofs that are not included in the version for publication.
7. Beeson, M., the Otter-λ website, temporarily to be found at http://mh215a.cs.sjsu.edu.
8. Boyer, R. S., and Moore, J. S., *A Computational Logic Handbook*, Academic Press, Boston (1988).
9. Robert Boyer, Ewing Lusk, William McCune, Ross Overbeek, Mark Stickel, and Lawrence Wos, Set theory in first-order logic: Clauses for Gödel's axioms. *Journal of Automated Reasoning* **2** 287–327, 1986.
10. Buchberger, B., *et. al.* A Survey of the *Theorema* Project, RISC Technical Report 97-15 (1997), available online at ftp://ftp.risc.uni-linz.ac.at/pub/techreports/1997/97-15.ps.gz.
11. Constable, R. L. *et. al.*, *Implementing Mathematics with the Nuprl Proof Development System*, Prentice-Hall, Englewood Cliffs, New Jersey (1986).
12. Coq development team, The Coq Proof Assistant Reference Manual: Version 7.2, Rapport Technique RT-0255 de l'INRIA (2002). www.inria.fr/rrrt/rt-0255.html
13. Coquand, T. and Huet, G., The Calculus of Constructions, *Information and Computation* **76**, 1988, 95-120.
14. Dowek, G., Higher-order unification and matching, Chapter 16 of[25].
15. 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).
16. Harrison, J., *Theorem Proving with the Real Numbers*, Springer-Verlag, Berlin/ Heidelberg/ New York (1998).
17. G. Huet, A unification algorithm for typed λ-calculus, *Theoretical Computer Science* **1** (1975) 27–52.
18. Lusk, E., McCune, W., Overbeek, R., Logic machine architecture: kernel functions, in: Loveland, D. W. (ed.) *6th Conference on Automated Deduction* 70–79, Springer-Verlag, Berlin/ Heidelberg/ New York (1982).
19. McCune, W., Otter 3.0 Reference Manual and Guide, Argonne National Laboratory Tech. Report ANL-94/6, 1994.
20. Miller, D., Cohen, E. L., and Andrews, P., A look at TPS, in: Loveland, D. (ed.), *6th*

Conference on Automated Deduction, 50–69, Lecture Notes in Computer Science 138, Springer-Verlag, (1982).

21. G. Nadathur and D. Miller, An Overview of λ-Prolog, in: Bowen and Kowalski (eds.), *Proceedings of the Fifth International Symposium on Logic Programming, Seattle, August 1988.*
22. Nipkow, T., Paulson, L. C., and Wenzel, M. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, Lecture Notes in Computer Science **2283**, Springer-Verlag, Berlin/ Heidelberg/ New York (2003).
23. Pietrzykowski, T., and Jensen, D., A complete mechanization of second order logic, *J. Assoc. Comp. Mach.* **20** (2) pp. 333-364, 1971. +
24. Pietrzykowski, T., and Jensen, D., A complete mechanization of ω-order type theory, *ASsoc. Comp. Math. Nat. Conf.* 1972, Vol. 1, 82–92.
25. 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.
26. S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. Srivas, PVS: Combining Specification, Proof Checking, and Model Checking, in Alur, R., and Henzinger, T. A. (eds.) *Computer-Aided Verification, CAV '96*, 411–414, Lecture Notes in Computer Science **1102**, Springer-Verlag, New Brunswick, N. J. (1996).
27. Trybulec, W. A. Subgroup and Cosets of Subgroups, *Journal of Formalized Mathematics* **12**, 1990.
    http://mizar.uwb.edu.pl/JFM/Vol2/group_2.miz.html.
28. Wick, C., and McCune, W., Automated reasoning about elementary point-set topology, *J. Automated Reasoning* **5(2)** 239–255, 1989.
29. Yuan Yu, Computer proofs in group theory, *J. Automated Reasoning* **6**(3) pp. 251–286, 1990.