Implicit and Explicit Typing in Lambda Logic

Michael Beeson[1]*

San José State University, San José, Calif.
beeson@cs.sjsu.edu,
www.cs.sjsu.edu/faculty/beeson

Abstract. Otter-lambda is a theorem-prover based on an untyped logic with lambda calculus, called lambda logic. Otter-lambda is built on Otter, so it uses resolution proof search, supplemented by demodulation and paramodulation for equality reasoning, but it also uses a new algorithm, lambda unification, to instantiate variables for functions or predicates. The idea of "implicit typing" is to "type" the function and predicate symbols by specifying the legal types of their arguments and return values. The hope is that if the axioms can be typed in this way then the consequences should be typeable too. This is true (with one restriction) in first-order logic. We show that by placing suitable restrictions on lambda unification, one can extend this theorem to lambda logic. All the interesting proofs obtained so far with Otter-lambda, except those explicitly involving untypeable axioms, are covered by this theorem. "Explicit typing" refers to the use of simple type-checking in addition to implicit typing.

1 Introduction

Lambda logic is an untyped system, and lambda unification is an untyped unification algorithm. Lambda logic is a consistent system with a completeness theorem [1], but the exact relationships between lambda logic and first order logic, and between lambda logic and typed logics, have still not been worked out, and some proofs in lambda logic seem at first glance surprisingly close to inconsistency. On the other hand, Otter-λ, which use lambda logic, has proved many interesting theorems whose proofs, by inspection, translate directly into first-order proofs. We may especially mention proofs by mathematical induction [2], where lambda unification is used to find the instance or instances of induction required; the proofs produced translate directly into formal Peano arithmetic PA, which has an infinite schema of first-order instances of induction, although Otter-λ works with a single axiom of induction with a variable for the predicate. We are interested in the following question: Suppose we have some axioms and a conjecture formulated in some typed logic. Suppose we erase the type labels, obtaining axioms and a conjecture in lambda logic, and run Otter-λ (or any other prover) and find a proof in lambda logic, using the usual first-order inference rules with lambda unification. Can we translate that proof back into a

* Research supported by NSF grant number CCR-0204362.

proof in the original logic? This is a delicate question: Otter-λ can produce some untypeable proofs, and it can also produce good, typeable proofs, in some cases from axioms which, when considered as untyped, are inconsistent.

This question already arises in first-order logic, without bringing in lambda unification to complicate the matter. The simplest case of "types" (when there are no "function types") is more commonly called "sorts". For example, we might have one "sort" for the natural numbers and another "sort" for elements of some ring. In that setting, consider the problem of proving that there are no nilpotent elements in an integral domain. An integral domain is a ring R in which $xy = 0$ implies $x = 0$ or $y = 0$, i.e. there are no zero divisors. A element c of R is called *nilpotent* if for some positive integer n, c^n is zero. Informally, one proves by induction on n that c^n is not zero. The equation defining exponentiation is $x^{s(n)} = x * x^n$. If c and c^n are both nonzero, then the integral domain axiom implies that c^{n+1} is also nonzero.

If we prove this theorem in a two-sorted logic (specifying the particular instance of induction to use, so that there is nothing "higher-order" about the problem) and then erase the sorts, and input the problem to a first-order theorem prover, and it finds a proof, do we know for certain in advance of inspecting the proof that we will be able to put back the "sorts" and construct a proof of the original theorem in two-sorted logic? This is a first-order example of the situation that concerns us. The answer in the first-order case [4] is that we can do it, as long as the first-order proof did not use paramodulation from or into variables. The answer for lambda logic is not quite so straightforward, since there do exist untypeable proofs in lambda logic. The "problem" is that in first-order resolution, variables get assigned a value in unification only when the variable occurs as an argument, either of a parent term or a parent literal, so there is only one possible type for that variable, if function and predicate symbols have unique types. But in lambda logic, that is not so, since we want to use a variable in a predicate or function position. (That is also the problem with paramodulation from variables: the type of a variable is not uniquely determined.)

The theorems in this paper start with a set of first-order clauses that could be the result of "erasing types": we assume that we have axioms and a conjecture that are "implicitly" typed (or typeable). We begin by defining this notion precisely. After that, we need to analyze the lambda unification algorithm, and see where mistyped terms might arise; or at least, under what general circumstances they do *not* arise. This leads to the notion of *type-safe lambda unification*. We then show that when we use type-safe lambda unification instead of ordinary unification in the usual first-order rules of inference, the same results on implicit typing that work for first-order logic continue to work for lambda logic.

What does this mean in practice? Suppose we start with an implicitly-typeable input file, and run Otter-λ. Then we have three possible ways to know that the proof is also typeable: (1) We could inspect the output proof and verify (by hand or machine) that it is indeed typeable; or (2) We could put the command `set(types)` in the input file, telling Otter-λ to use certain restrictions that guarantee only type-safe lambda unifications will be made, and appeal to

the theorems about implicit typing in this paper; or (3) we could put into the input file some explicit typings for the function, predicate, and constant symbols, under `list(types)`. For example, $type(N, s(N))$ says that s takes integers to integers; $type(R, pow(R, N))$ says that $pow(x, n)$ has type R if x has type R and n has type N. If `list(types)` is present, then Otter-λ respects these typings in performing lambda unification; and again, by the theorems of this paper, any proof that Otter-λ finds with `list(types)` present is guaranteed to be typeable.

2 Implicit typing in first order logic

We begin with the "no-nilpotents" example. To formalize this theorem in first order logic we might use two unary predicates $R(x)$ and $N(x)$, whose meaning would be "x is a member of the ring R" and "x is a natural number", respectively. Then the ring axioms would be "relativized to R", which means that instead of saying $x + 0 = 0$, we would say $R(x) \rightarrow x + 0 = 0$, or in clausal form, $-R(x)|x + 0 = 0$. (The vertical bar means "or", and the minus sign means "not".) Similarly, the axiom of induction would be relativized to N. The axiom of induction is usually formulated using a symbol s for the successor function, or "next-integer" function. For example, $s(4) = 5$. The specific instance of induction we need for this proof can be expressed by the two (unrelativized) clauses

$$x^o \neq 0 \mid x^{g(x)} = 0 \mid x^n = 0.$$
$$x \not= 0 \mid x^{s(g(x))} \neq 0 \mid x^n = 0.$$

To see that this expresses induction, think of $g(x)$ as a constant (on which x is not allowed to depend). Then the middle literal of the first clause is $x^c = 0$. That is the induction hypothesis. The middle literal of the second clause is $x^{s(c)} \neq 0$. That is the negated conclusion of the induction step. We have used o instead of 0 for the natural number zero, perhaps not the same as the ring element 0.[1] Now here is the question: when formalizing this problem, do we need to relativize the axioms using $R(x)$ and $N(x)$, or not? Experimentally, if we put the unrelativized axioms into Otter, we do find a proof. Certainly this proof shows that in any integral domain whose underlying set is the natural numbers, there are no nilpotents, since in that case all the variables range over the same set, and no question of typing arises. But that is not the theorem that we set out to prove, so it may appear that we must use $R(x)$, $N(x)$, and relativization to formalize this problem. That is, however, not so. The method of "implicit typing" shows that under certain circumstances we can dispense with

[1] The relativized versions of the induction axioms would be

$$-R(x) \mid -N(n) \mid x^o \neq 0 \mid x^{g(x,n)} = 0 \mid x^n = 0.$$
$$-R(x) \mid -N(n) \mid x^o \neq 0 \mid x^{s(g(x,n))} \neq 0 \mid x^n = 0.$$
$$-R(x) \mid -N(n) \mid N(g(n, x)).$$

unary predicates such as R and N. Each argument position of each function or predicate symbol is assigned a type and the symbol is also assigned a "value type" or "return type". For example, in this problem the ring operations $+$ and $*$ have the type of functions taking two R arguments and producing an R value, 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 $type(N, \text{plus}(N, N))$, and we need to use a different symbol for 0 in the ring and zero in N. The Skolem symbol g in the induction axiom has the type specification $type(N, g(R))$. The exponentiation function has the type specification $type(R, R^N))$. The value type of predicate symbols must be *Prop*.

Constants are considered as 0-ary function symbols, so they are assigned types, for example $type(R, 0)$ and $type(N, o)$. 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. Variables are not assigned types. Instead, 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) 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 in [4].

Definition 1. *A* type specification *is an expression of the form* $type(R, f(U, V))$, *where* R, U, *and* V *are "type symbols". Any first-order terms not containing variables may be used as type symbols. Here 'type' must occur literally, and* f *can be any symbol. The number of arguments of* f, *here shown as two, can be any number, including zero.*

The type R is called the *value type* of f. The symbol f is called the symbol of the type specification, and the number of arguments of f is the *arity*.

Definition 2. *A* typing *of a term is an assignment of types to the variables occurring in the term and to each subterm of the term. A typing of a literal is similar, but the formula itself must get value type Prop. A typing of a clause is an assignment that simultaneously types all the literals of the clause. A typing of a term (or literal or clause or set of clauses)* t *is* correct with respect to a list of type specifications S *provided that*

(i) each occurrence of a variable in t *is assigned the same type.*

(ii) each subterm r *of* t *is typed according to a type specification in* S. *That is, if* r *is* $f(u, v)$ *and* $f(u, v)$,u, *and* v *are assigned types* a, b, *and* c *respectively, then there is a type specification in* S *of the form* $type(a, f(b, c))$.

(iii) each occurrence of each subterm r *of* t *in* t *has the same value type.*

In the definition, nothing prevents S from having more than one type specification for the same function symbol and arity. Condition (iii) is needed in such a case. The phrase, *correctly typed term* t, is short for "term t and a correct typing of t with respect to some list of type specifications given by the context". The simplest theorem on implicit typing concerns (binary) resolution.[2]

Theorem 1. *Suppose each function symbol and constant occurring in a theory T is assigned a unique type specification, in such a way that all the axioms of T are correctly typed (with respect to this list of type specifications). Then conclusions reached from T by binary resolution (using first-order unification) are also correctly typed.*

Remark. This theorem is perhaps implicit in [4]. It is a special case of the theorem for lambda logic proved below, so we do not present a separate proof.

Does this theorem apply to the no-nilpotents example? We have to be careful about the type specification of the equality symbol. If we specify $type(\text{bool}, = (R, R))$, then we cannot use the same equality symbol in the axioms for the natural numbers, for example $s(x) \neq 0$ and $x = y \mid s(x) \neq s(y)$. However, Otter treats any symbol beginning with EQ as an equality; = is a synonym for EQ, but one can also use, for example EQ2. Therefore, if we want to apply the theorem, we need to use two different equality symbols.

The theorem above can be extended to include the additional rules of inference factoring, paramodulation, and demodulation. [3]

Theorem 2. *Suppose each function symbol and constant occurring in a theory T is assigned a unique type specification, in such a way that all the axioms of T are correctly typed (with respect to this list of type specifications). The type specifications of equality symbols must have the form $type(\text{bool}, = (X, X))$ for some type X. Then conclusions reached from T by binary resolution, hyperresolution, factoring, demodulation, and paramodulation (using first-order unification in applying these rules) are also correctly typed, provided demodulation and paramodulation are not applied to or from variables.*

[2] In the following theorem, we assume (as is customary with resolution) that after a theory has been brought to clausal form, the variables in distinct clauses are renamed so that no variable occurs in more than one clause.

[3] For those not familiar with those rules we review their definitions. *Factoring* permits the derivation of a new clause by unifying two literals in the same clause that have the same sign, and applying the resulting substitution to the entire clause. *Paramodulation* is the following: suppose we have already deduced $t = q$ (or $q = t$) and $P[z := r]$, and unification of t and r produces a substitution σ such that $t\sigma = r\sigma$; then we can deduce $P[z := q\sigma]$. Paramodulation *from variables* is the case in which t is a variable. Paramodulation *into a variable* is the case in which r is a variable. Demodulation is similar to paramodulation, except that (i) unlike paramodulation, it is unidirectional (i.e., the hypothesis must be $t = q$, not $q = t$), (ii) it is applied only under certain circumstances and using formulas designated in an input file as "demodulators". From the point of view of soundness proofs, it is a special case of paramodulation.

Proof. This is a special case of Theorem 4 below, which treats lambda logic.

Example. One cannot allow "overloading", or multiple type specifications for the same symbol, and still use implicit typing correctly. Suppose we want to use $x + y$ both for natural numbers and for integers. Thinking of integers, we write the axiom $x + (-x) = 0$, and thinking of natural numbers we write $1 + x \neq 0$. Resolving these clauses, we find a contradiction upon taking $x = 1$.

Example. This example illuminates the situation with regard to paramodulation from variables. Consider the three unit clauses $x = a$, $P(b)$, and $-P(c)$. These clauses lead to a contradiction using paramodulation from the variable x and binary resolution. But without paramodulation from variables, no contradiction can be derived. This shows that we have lost first-order refutation completeness, already in the first order case, as the price of implicit typing. But this is good: if equality is between objects of type A and P is a predicate on objects of type B, then these clauses are not contradictory. This loss of first-order completeness already occurs in the first-order case, and is not a phenomenon special to lambda logic. *Question*: "but if b and c have the same type, then shouldn't the contradiction be found?" *Answer*: 'b' and 'c' are constants in an untyped language, so they do not have types. What the example shows is that, if many-sorted *models* are considered, there are models of this theory, even though the theory has no first-order models.

3 Lambda logic and lambda unification

Lambda logic is the logical system one obtains by adding lambda calculus to first order logic. This system is formulated, and some fundamental metatheorems are proved, in [1]. 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. An algorithm for producing lambda unifying substitutions, called *lambda unification*, is used in the theorem prover Otter-λ, which is based on lambda logic rather than first-order logic, but is built on the well-known first-order prover Otter [3]. 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 $Ap(x, y)$ for x applied to y, which is often abbreviated in technical papers to $x(y)$ or even xy. In this paper, Ap and $lambda$ will always be written explicitly.

Our main objective in this section is to define the lambda unification algorithm. This is a non-deterministic algorithm: it can return, in general, many different unifying substitutions for two given input terms. As for ordinary unification, the input is two terms t and s (this time terms of lambda logic) and 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 $g(s_1, \ldots, s_m)$, this clause does not apply unless $f = g$ and $n = m$; in that case we do the following:

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

The rule related to alpha-conversion says that, if we want to unify $lambda(z, t)$ with $lambda(x, s)$, let τ be the substitution $z := x$ and then unify $t\tau$ with s, rejecting any substitution that assigns a value depending on x.[4] 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, may not 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.

There are two sources of non-determinism in the above, namely steps 1 and 3.[5]

[4] 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.

[5] Step 1 is made deterministic in Otter-λ as follows: in step 1, if x occurs in t, we pick the largest masking subterm q that occurs as a second argument of Ap. The point of this choice is that, if we want the proof to be implicitly typeable, then q should be chosen to have the same type as w, and w is a second argument of Ap. If x occurs in t, but no masking subterm occurs as a second argument of Ap, we pick the

Finally, lambda unification needs a rule 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:

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

Example. Lambda unification can lead to untypeable proofs, for example those needed to produce fixed points in lambda calculus. As an example, if we unify $Ap(x, y)$ with $f(Ap(x, y))$, the masking subterm q is x itself; w is y so σ is $y := x$; $w\sigma$ is x and $t\sigma$ is $Ap(x, x)$. Thus we get the following result:[6]

$$x := lambda(z, f(Ap(z, z))) \qquad\qquad y := x$$

Type restrictions will be violated if we have specified the typing:

$$type(B, Ap(i(A, B), A)). \qquad\qquad type(B, f(B)).$$

Variable x has type $i(A, B)$, and variable y has type A, so the unification of x and y violates type restrictions, since $i(A, B)$ is not the same type as A.

Definition 3. *We say that a particular lambda unification (of $Ap(X, w)$ with t) is* type-safe *(with respect to some explicit or implicit typings) if the masking subterm q selected by lambda unification has the same type (with respect to those typings) as the term w, and q is a proper subterm of t (unless the two arguments of Ap have the same type). We also require that the value type assigned to $Ap(X, w)$ is the same as the value type assigned to t.*

---

smallest masking subterm. If x does not occur in t, we pick a constant that occurs in t; if there is none, we fail. 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 it is important to backtrack over multiple choices in this step. Early versions of Otter-λ made a deterministic choice, but since December 2005, Otter-λ can backtrack over different choices of S, returning up to a pre-specified number `max_unifiers` of different unifiers. Our proofs in this paper apply to the full non-deterministic lambda unification, as well as to any versions obtained by restricting the choices of possible masking terms.

[6] The symbol i does not have to be "defined" here; type assignments can be arbitrary terms. But intuitively, $i(A, B)$ could be thought of as the type of functions from type A to type B.

The example preceding the definition illustrates a lambda unification that is not type-safe for *any* reasonable typing. The masking subterm is x; type safety would require x to be assigned the same type as y. But x occurs as a first argument of Ap and y as a second argument of Ap. Therefore the type specification of Ap would have to be of the form $type(V, Ap(U, U))$; but normally Ap will have a type specification of the form $type(B, Ap(i(A, B), A))$.

4 Implicit typing in lambda logic

In lambda logic, we can state the axiom of mathematical induction in full generality, and Otter-λ can use lambda unification to find the specific instance of induction that is required. The proof, many of which are exhibited in [2] are correctly typeable. We will show that this is not an accident.

Definition 4. *A list of type specifications S is called* coherent *if*
(1) for each (predicate or function) symbol f (except possibly Ap and $lambda$) and arity n, it contains at most one type specification of symbol f and arity n; the value type of a predicate symbol must be $Prop$ and of a function symbol, must not be $Prop$.

(2) $type(i(X, Y), lambda(X, Y))$ belongs to S if and only if $type(Y, Ap(i(X, Y), X))$ belongs to S.

(3) all type specifications with symbol Ap have the form $type(V, Ap(i(U, V), U))$, for the same type U, which is called the "ground type" of S.

(4) all type specifications with symbol $lambda$ have the form $type(i(U, V), lambda(U, V))$,[7] where U is the ground type of S.

(5) There are at most two type specifications in S with symbol Ap; if there are two, then exactly one must have value type $Prop$.

Conditions (2) and (3) guarantee that beta-reduction carries correctly typed terms to correctly typed terms. One might wish for a less restrictive condition in (4) and (5), allowing functions of functions, or functions of functions of functions, etc. But this is the condition for which we can prove theorems at the present time, and it covers a number of interesting examples in algebra and number theory.

If S is a coherent list S of type specifications, it makes sense to speak of "the type assigned to a term t by S", if there is at least one type specification in S for the main symbol and arity of t. Namely, unless the main symbol of t is Ap, only one specification in S can apply, and if the main symbol of t is Ap, then we apply the specification that does not have value type $Prop$. Similarly, it makes sense to speak of "the type assigned to an atomic formula by S". When the main symbol of t is Ap, we can speak of "the type assigned to t as a term" or "the type assigned to t as a formula", using the specification that does not or does have $Prop$ for its value type.

[7] Intuitively, this says that if z has type X and t has type Y then $lambda(z, t)$ has type $i(X, Y)$, the type of functions from X to Y.

Theorem 3. *Let S be a coherent list of type specifications. Let s and t be two correctly typed terms or two correctly typed atomic formulas with respect to S. Let σ be a substitution produced by successful type-safe lambda unification of s and t. Then $s\sigma$ and $t\sigma$ are correctly typed, and S assigns the same type to s, t, and $s\sigma$.*

Example. Let s be $Ap(X, w)$ and t be $a+b$. We can unify s and t by the substitution σ given by $X := lambda(x, x + b)$ and $w := a$. If $type(0, Ap(i(0, 0), 0))$ and $type(0, +(0, 0))$ then these are correctly typed terms and the types of $s\sigma$ and $a+b$ are both 0. Perhaps Ap also has a type specification $type(Prop, Ap(i(0, Prop), 0))$, used when the first argument of Ap defines a propositional function. However, this additional type specification will not lead to mis-typed unifications, since the two type specifications of Ap are coherent.

Proof. We proceed by induction on the length of the computation by lambda unification of the substitution σ.

(i) Suppose s is a term $f(r, q)$ (or with more arguments to f), and either f is not Ap, or r is neither a variable nor a lambda term. Then t also as the form $f(R, Q)$ for some R and Q, and σ is the result of unifying r with R to get $r\tau = R\tau$ and then unifying $q\tau$ with $Q\tau$, producing substitution ρ so that $\sigma = \tau \circ \rho$. By the induction hypothesis, $r\tau$ is correctly typed and gets the same type as r and $R\tau$; again by the induction hypothesis, $q\tau\rho$ and $Q\tau\rho$ are correctly typed and get the same type as q. Then $s\sigma = f(r\sigma, q\sigma) = f(r\tau\rho, q\tau\rho)$ is also correctly typed.

(ii) The argument in (i) also applies if s is $Ap(r, q)$ and t is $Ap(R, Q)$ and lambda unification succeeds by unifying these terms as if they were first-order terms.

(iii) If s is a constant, then $s\sigma$ is s and there is nothing to prove.

(iv) If s is a variable, what must be proved is that t and s have the same value type. A variable must occur as an argument of some term (or atom) and hence the situation really is that we are unifying $P(s, \ldots)$ with some term q, where P is either a function symbol or a predicate symbol. If P is not Ap, then q must have the form $P(t, \ldots)$, and t and s occur in corresponding argument positions (not necessarily the first as shown). Since these terms or atoms $P(t, \ldots)$ and $P(s, \ldots)$ are correctly typed, and S is coherent, t and s do have the same types. The case when P is Ap will be treated below.

(v) Suppose s is $Ap(r, q)$, where $r = lambda(z, p)$, and z does occur in p. Then s beta-reduces to $p[z := q]$, and lambda unification is called recursively to unify $p[z := q]$ with t. By induction hypothesis, t, $t\sigma$, $p[z := q]$, and $p[z := q]\sigma$ are well-typed and are assigned the same value type, which must be the value type, say V, of p. Since S is coherent, the type assigned to $lambda(z, p)$ is $i(U, V)$, where U is the "ground type", the type of the second argument of Ap. The type of q is U since q occurs as the second arg of Ap in the well-typed term s. The type of s, which is $Ap(r, q)$, is V. We must show that $s\sigma$ is well-typed and assigned the value type V. Now $s\sigma$ is $Ap(r\sigma, q\sigma)$. It suffices to show that $q\sigma$ has type U and $r\sigma$ has type $i(U, V)$. We first show that the type of $q\sigma$ is U. Since z has type U in $lambda(z, p)$, $q\sigma$ occurs in the same argument positions in $p[z := q]\sigma$ as z

does in p, and since z does occur at least once in p, and $p[z := q]\sigma$ is well-typed, $q\sigma$ must have the same type as z, namely U. Next we will show that $r\sigma$ has type $i(U, V)$. We have $r\sigma = lambda(z, p)\sigma = lambda(z, p\sigma)$ (since the bound variable z is not in the domain of σ). We have $p\sigma[z := q\sigma] = p[z := q]\sigma]$ and the type of the latter term is V as shown above. The type of $A[z := B]$ is the type of A, and moreover $A[z := B]$ is well-typed provided A and B are well-typed and z gets the same type as B. That observation applies here with $A = p\sigma$ and $B = q\sigma$, since the type of z is U and the type of $q\sigma$ is U. Therefore the type of $p\sigma$ is the same as the type of $p\sigma[z := q\sigma]$, which is the same as $p[z := q]\sigma$, which has type the same as $p[z := q]$, which we showed above to be V. Since $r\sigma = lambda(z, p\sigma)$, and z has type U, $r\sigma$ has type $i(U, V)$, which was what had to be proved.

(vi) Suppose s is $Ap(r, w)$ and r is not a variable. Then we create a fresh variable X, unify $Ap(X, w)$ with t generating substitution σ. By induction hypothesis, $t\sigma$ has the same type as $Ap(X, w)$. Then we unify $X\sigma$ with $r\sigma$, generating substitution τ; by induction hypothesis, the types of $X\sigma$ and $r\sigma$ agree. The result of the unification is $\sigma\tau$. We have to check that $Ap(r, w)\sigma\tau$ has the same type as $t\sigma\tau$. But $Ap(r, w)\sigma\tau = Ap(r\sigma, w\sigma)\tau$; since $r\sigma$ and $X\sigma$ have the same type, $Ap(r, w)\sigma\tau$ has the same type as $Ap(X\sigma, w\sigma)\tau = Ap(X, w)\sigma\tau = t\sigma\tau$.

(vii) There are two cases not yet treated: when s is $Ap(X, w)$, and when s is a variable X occurring in the context $Ap(X, w)$. We will treat these cases simultaneously. As described in the previous section, the algorithm will (1) select a masking subterm $q\sigma$ of $t\sigma$ (2) unify w and q with result σ (failing if this fails), (3) create a new variable z, and substitute z for some or all occurrences of $q\sigma$ in $t\sigma$, obtaining r, and (4) produce the unifying substitution σ together with $X := lambda(z, r)$.

Assume that t is a correctly typed term. Then every occurrence of q in t has the same type, by the definition of correctly typed. Since by hypothesis this is type-safe lambda unification, q and w have the same type, call it U. Since q unifies with w, by the induction hypothesis $q\sigma$ and $w\sigma$ are correctly typed and get the same types as q and w, respectively, namely U. If $Ap(X, w)$ has type $Prop$, then the type of s and that of t are the same by hypothesis. Otherwise, both occur as arguments of some function or predicate symbol P, in corresponding argument positions, and hence, by the coherence of S, they are assigned the same (value) type V. Then X has the type $i(U, V)$. We now assign the fresh variable z the type U; then r is also correctly typed, and gets the same type V as s and t, since it is obtained by substituting z for some occurrences of $q\sigma$ in $t\sigma$. For this last conclusion we need to use the fact that q is a proper subterm of t, by the definition of type-safe unification; hence r is not a variable, so the value type of r is well-defined, since S is coherent. Since S is coherent, there is a type specification in S of the form $type(i(U, V), lambda(U, V))$. Thus the term $lambda(z, r)$ can be correctly typed with type $i(U, V)$, the same type as X. Hence $X\sigma$ has the same type as X, and $s\sigma$ has the same type as s. That completes the proof of the theorem.

Theorem 4 (Implicit Typing for Lambda Logic). *Let A be a set of clauses, and let S be a coherent set of type specifications such that each clause in A is correctly typeable with respect to S. Then all conclusions derived from A by binary resolution, hyperresolution, factoring, paramodulation, and demodulation (including beta-reduction), using type-safe lambda unification in these rules of inference, are correctly typeable with respect to S, provided paramodulation from or into variables or Ap terms is not allowed, and demodulators are not allowed to have variables or Ap terms on the left.*

Example. To show that the second restriction on paramodulation is necessary: Suppose Ap has a type specification $type(Prop, Ap(i(0, Prop), 0))$. Without the restriction, we could paramodulate from $x + 0 = x$ into $Ap(X, x)$, unifying $x + 0$ with $Ap(X, x)$ as in the example after Theorem 3, with the substitution $X := lambda(x, x + 0)$. The conclusion of the paramodulation inference would be x. That is a mistyped conclusion, since x does not have the type $Prop$, although Ap does have value type $Prop$.

Proof. Note that a typing assigns type symbols to variables, and the scope of a variable is the clause in which it occurs, so as usual with resolution, we assume that all the variables are renamed, or indexed with clause numbers, or otherwise made distinct, so that the same variable cannot occur in different clauses. In that case the originally separate correct typings $T[i]$ (each obtained from S by assigning values to variables in clause $C[i]$) can be combined (by union of their graphs) into a single typing T. We claim that the set of clauses A is correctly typed with respect to this typing T. To prove this correctness we need to prove:

(i) *each occurrence of a variable in A is assigned the same type by T.* This follows from the correctness of $C[i]$, since because the variables have been renamed, all occurrences of any given variable are contained in a single clause $C[i]$.

(ii) *If r is $f(u, v)$, and r occurs in A, and $f(u, v)$,u, and v are assigned types a,b,c respectively, then there is a type specification in S of the form $type(a, f(b, c))$.* If the term r occurs in A, then r occurs in some $C[i]$, so by the correctness of $T[i]$, there is a type specification in S as required.

(iii) *each occurrence of each term r that occurs in A has the same value type.* This follows from the coherence of S. The different typings $T[i]$ are not allowed to assign different value types to the same symbol and arity.

Hence A is correctly typed with respect to T.

All references to correct typing in the rest of the proof refer to the typing T.

We prove by induction on the length of proofs that all proofs from A using the specified rules of inference lead to correctly typed conclusions. The base case of the induction is just the hypothesis that A is correctly typeable. For the induction step, we take the rules of inference one at a time. We begin with binary resolution. Suppose the two clauses being resolved are $P|Q$ and $-R|B$, where substitution σ is produced by lambda unification and satisfies $P\sigma = R\sigma$. Here Q and B can stand for lists of more than one literal, in other words the rest of the literals in the clause, and the fact that we have shown P and $-R$ as the first literals in the clause is for notational convenience only. By hypothesis, $P|Q$ is correctly typed with respect to S, and so is $-R|B$, and by Theorem

3, $P\sigma|Q\sigma$ and $-R\sigma|B\sigma$ are also correctly typed. The result of the inference is $Q\sigma|B\sigma$. But the union of correctly typed terms, literals, or sets of literals (with respect to a coherent set of type specifications) is again correctly typed, by the same argument as in the first part of the proof. In other words, coherence implies that if some subterm r occurs in both $Q\sigma$ and in $B\sigma$ then r gets the same value type in both occurrences. That completes the induction step when the rule of inference is binary resolution.

Hyperresolution and negative hyperresolution can be "simulated" by a sequence of binary resolutions, so the case in which the rule of inference is hyperresolution or negative hyperresolution reduces to the case of binary resolution. The rule of "factoring" permits the derivation of a new clause by unifying two literals in the same clause that have the same sign, and applying the resulting substitution to the entire clause. By Theorem 3, a clause derived in this way is well-typed if its premise is well-typed.

Now consider paramodulation. In that case we have already deduced $t = q$ and $P[z := r]$, and unification of t and r produces a substitution σ such that $t\sigma = r\sigma$. The conclusion of the rule is $P[z := q\sigma]$. We have disallowed paramodulation from or into variables or Ap terms in the statement of the theorem; therefore t and r are not variables or Ap terms. Let us write $Type(t)$ for the value type of (any term) t. Because $t = q$ is correctly typed, we have $Type(t) = Type(q)$. Since neither t nor q is an Ap term, then they have the same functor, and hence $Type(t\sigma) = Type(q\sigma)$. Then by Theorem 3, $Type(t\sigma) = Type(t)$ and $Type(q\sigma) = Type(q) = Type(t) = Type(t\sigma)$. Thus in any case $Type(q\sigma) = Type(t\sigma)$. The value type of r is the same at every occurrence, since $P[z := r]$ is correctly typed. To show that $P[z := q\sigma]$ is correctly typed, it suffices to show that $Type(q\sigma) = Type(r)$, which is the same as the type of $r\sigma$. Since the terms t and r unify, and neither is a variable, their main symbols are the same, since by hypothesis r is not of the form $Ap(X, w)$, with X a variable or functional term. Hence $Type(r) = Type(r\sigma) = Type(t\sigma) = Type(q\sigma)$, which is what had to be shown.

Now consider demodulation. In this case we have already deduced $t = q$ and $P[z := t\sigma]$ and we conclude $P[z := q\sigma]$, where the substitution σ is produced by lambda unification of t with some subterm ρ of $P[z := \rho]$. Taking $r = t\sigma$, we see that demodulation is a special case of paramodulation, so we have already proved what is required. That completes the proof of the theorem.

Example: fixed points. There is a fixed point argument that shows that the (unrelativized) group axioms are contradictory in lambda logic. Briefly, we construct g such that $Ap(g, x) = c * Ap(g, x)$; it follows that c is the group identity, so there is only one object, a contradiction in lambda logic.[8] The fixed point is constructed using a term $Ap(f, Ap(x, x))$. The part of this that is problematic is $Ap(x, x)$. If the type specification for Ap is $type(V, Ap(i(U, V), U))$, then for $Ap(x, x)$ to be correctly typed, we must have $V = U = i(U, U)$. If U and V are type symbols, this can never happen, so the fixed point construction cannot be correctly typed. It follows from the theorem above that this argument cannot be

[8] Semantically, this means that you cannot make a lambda model into a group.

found by Otter-λ from a correctly typed input file. In particular, in an input file containing correctly typed axioms, we will not get a contradiction from a fixed point argument.

On the other hand, in file `lambda4.in`, we show that Otter-λ can verify the fixed-point construction. The input file contains the negated goal

$$\begin{aligned} & Ap(c, Ap(lambda(x, Ap(c, Ap(x, x))), lambda(x, Ap(c, Ap(x, x))))) \\ \neq\ & Ap(lambda(x, Ap(c, Ap(x, x))), lambda(x, Ap(c, Ap(x, x)))). \end{aligned}$$

Since this contains the term $Ap(x, x)$, it cannot be correctly typed with respect to any coherent list of type specifications T. Otter-λ does find a proof using this input file, which is consistent with our argument above that fixed-point constructions will not occur in proofs from correctly typeable input files. The fact that the input file cannot be correctly typed, which we just observed directly, can also be seen as a corollary of the theorem, since Otter-λ finds a proof.

To summarize: (1) The (unrelativized) axioms of group theory are contradictory in lambda logic, but if we put in only correctly-typed axioms, Otter-λ will find only correctly typed proofs, which will be valid in the finite type structure based on any group, and hence will not be proofs of a contradiction. (2) We already knew that resolution plus factoring plus paramodulation from non-variables is not refutation-complete, even for first-order logic; and we remarked when pointing that out that this permits typed models of some theories that are inconsistent when every object must have the same type. Here is another illustration of that phenomenon in the context of lambda logic. (3) Of course Otter-λ *can* find the fixed-point proof that gives the contradiction; but to make it do so, we need to put in some non-well-typed axiom, such as the negation of the fixed-point equation.

5 Enforcing type-safety

The theorems above are formulated in the abstract, rather than being theorems about a particular implementation of a particular theorem-prover. As a practical matter, we wish to formulate a theorem that does apply to Otter-λ and covers the examples posted on the Otter-λ website, some of which have been mentioned here. Otter-λ never uses paramodulation into or from variables, so that hypothesis of the above theorems is always satisfied. But Otter-λ does not always use only type-safe lambda unification; nor would we want it to do so, since it can find some untyped proofs of interest, e.g. fixed points, Russell's paradox, etc. We have two ways of restricting Otter-λ: *implicitly*, by putting `set(types)` in the input file, or explicitly, by putting a list of *explicit* type specifications in the input file. This command `set(types)` causes Otter-λ to use *restricted lambda unification*. That means that, when selecting a masking subterm, only a second argument of Ap or a constant will be chosen. We prove that this enforces type safety under certain conditions:

Theorem 5 (Type safety of restricted lambda unification). *Suppose that a given set of axioms admits a coherent type specification in which there is no typing of the form $Ap(U,U)$, and all constants receive type U. Then all deductions from the given axioms by binary resolution, factoring, hyperresolution, demodulation (including beta-reduction) paramodulation (except into or from variables and Ap terms), lead to correctly typeable conclusions, provided that restricted lambda unification is used in those rules of inference.*

Proof. It suffices to show that lambda unifications will be type-safe under these hypotheses. The unification of $Ap(x,w)$ with t is type-safe (by definition) if in step (1) of the definition of lambda unification, the masking subterm q of t has the same type as w. Now q is either a constant or term containing x that appears as a second argument of Ap, since those are the "restrictions" in restricted lambda unification. If q is a variable then it must be x, and must occur as a second argument of Ap; but x occurs as a first argument of Ap, and all second arguments of Ap get the same type, so there must be a typing of the form $type(T, Ap(U,U))$. But such a typing is not allowed, by hypothesis. Therefore q is not a variable. Then if q contains x, it must occur as a second argument of Ap, as does w; hence by hypothesis w and q get the same type. Hence we may assume q is a constant. But by hypothesis, all constants get the same type as the second arguments of Ap. That completes the proof.

Examples. The proofs by induction in [2] fulfill the hypotheses of this theorem, and hence we are justified in not relativizing the induction axiom to N. The nonilpotents example appears *prima facie* not to meet the hypotheses of Theorem 5, since that theorem requires that all constants have the same type as the second argument of Ap, in this case N, but we have a constant o of type R. This is not a serious problem; it can be solved either by implicit typing or by explicit typing. To solve it by implicit typing, we replace o in the axioms by $zero(0)$, where $zero$ is a new function symbol with the type specification $type(R, zero(N))$.

References

1. 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).
2. Beeson, M., Mathematical induction in Otter-λ, accepted for publication in *J. Automated Reasoning*, to appear in 2006. Available on the author's website.
3. McCune, W., Otter 3.0 Reference Manual and Guide, Argonne National Laboratory Tech. Report ANL-94/6, 1994.
4. Wick, C., and McCune, W., Automated reasoning about elementary point-set topology, *J. Automated Reasoning* **5(2)** 239–255, 1989.