Implicit 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, for instantiating variables for functions or predicates. The basic idea of a typed interpretation of a proof is to "type" the function and predicate symbols by specifying the legal types of their arguments and return values. The idea of "implicit typing" is that if the axioms can be typed in this way then the consequences should be typable too. This is not true in general if unrestricted lambda unification is allowed, but for a restricted form of "type-safe" lambda unification it is true. The main theorem of the paper shows that the ability to type proofs if the axioms can be typed works for the rules of inference used by Otter-lambda, if type-safe lambda unification is used, and if demodulation and paramodulation from or into variables are not allowed. All the interesting proofs obtained with Otter-lambda, except those explicitly involving untypable constructions such as fixed-points, are covered by this theorem.

1 Introduction: the no-nilpotents example

We begin with an example. Consider the problem of proving that there are no nilpotent elements in an integral domain. To explain the problem: 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 (i.e., c multiplied by itself n times) 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. It is a very simple proof, but it is interesting because it involves two *types* of objects, ring elements and natural numbers, and the proof involves a mix of the algebraic axioms and the number-theoretical axioms (mathematical induction). Since the proof is so simple, we can consider the issues raised by having two types of objects without being distracted by a complicated proof.

How are we to formalize this theorem in first order logic? The traditional way would be to have 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

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

To see that this corresponds to 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, which might not be the same as the ring element 0.

A traditional course in logic would teach you that to formalize this problem, you need to relativize all the axioms using R and N. Just to be explicit, the relativized versions of the induction axioms would be

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

and we would need additional axioms such as these:

$$-R(x) \mid -N(n) \mid R(x^n).$$
$$-R(x) \mid -R(y) \mid R(x + y).$$
$$-R(x) \mid -R(y) \mid R(x * y).$$
$$-R(x) \mid x + 0 = 0.$$

and so on for the other ring axioms.

2 Implicit typing in first order logic

Now here is the question: when formalizing this problem, do we need to relativize the induction axioms and the ring axioms using $R(x)$ and $N(x)$, or not? Experimentally, if we put the unrelativized axioms into Otter (Otter-λ is not needed, since we have explicitly given the prover the required instance of induction), we do find a proof. What does this proof actually prove? Certainly it 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. We can prove informally that any countable integral domain is isomorphic to one whose underlying set is the natural numbers. But this 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 unary predicates such as R and N. 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 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 $type(N, \mathrm{plus}(N, N))$, 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 $type(R, R^N))$.

Constants are considered as 0-ary function symbols, so they get 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. Types as we discuss them here are not quite the same as types in most programming languages, where variables are declared to have a certain type. Here, 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 long ago in [4], but McCune says it was already folklore at that time. It implies that the proof Otter finds using an input file without relativization actually is a valid proof of the theorem, rather than just of the special case where the ring elements are the natural numbers.

"Implicit typing" is the name of this technique, in which unary predicates whose function would be to establish typing are omitted. There are two ways to use implicit typing. First, we could just omit the unary predicates, let a theorem-proving program find a proof, and afterwards verify by hand (or by a computer program) that the proof is indeed well-typed. Second, we could verify that the axioms are well-typed, and prove that the inference rules used in the prover lead from correctly typed clauses to correctly typed clauses. Let us explore this second alternative. In order to state and prove a theorem, we first give some definitions:

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* bool. *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".

Remark. Allowing type specifications to contains variables would correspond to polymorphic types, i.e. overloading of function symbols. We do not allow such typings, but of course at the meta-level we can refer to a "typing of the form $i(U, U)$." That covers any specific typing such as $i(N, N)$, etc. For first-order theories, usually constant terms will suffice for naming the types (which are then usually called *sorts* rather than types, as in "multi-sorted logic").

The simplest theorem on implicit typing concerns the inference rule of (binary) resolution.[1]

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]. We give it here mainly to prepare the way for extensions to lambda logic in the next section.

[1] 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.

Proof. Suppose that literal $P(r)$ resolves with literal $-P(t)$, where r and t are terms; then there is a substitution σ such that $r\sigma = t\sigma$, the unifying substitution. Here P stands for any atomic formula and t and r might stand for several terms if P has more than one argument position. Since $P(r)$ and $P(t)$ are correctly typed by hypothesis, r and t must have the same value type (if they are not variables). The result of the resolution will be a disjunction of literals $Q\sigma|S\sigma$, where Q and S are the remaining (unresolved) literals in the clauses that originally contained $P(r)$ and $-P(t)$, respectively. Now Q and S are correctly typed by hypothesis, so we just need to show that applying the substitution σ to a correctly typed term or literal will produce a correctly typed term or literal. This will be true by induction on the complexity of terms, provided that substitution σ assigns to each variable x in its domain, a term q whose value type is the same as the value type of x in the clause in which x occurs. In first-order unification (but not in lambda unification) variables get assigned a value in unification only when the variable occurs as an argument, either of a parent term or a parent literal. That is, a variable cannot occur in the position of a literal. Thus when we are unifying $f(x, u)$ and $f(q, v)$, x will get assigned to q, and the type of x and the value type of q must be the same since they are both in the first argument place of f. That completes the 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 $x = y|s(x) \neq s(y)$. 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. Of course, we could just use = throughout and verify afterwards that the proof can be correctly typed, as = is never used in the same clause for equality between natural numbers and equality between ring elements; but if we want to be assured in advance that any proof Otter will find will be correctly typable, then we need to use different equality symbols. If we do so, then the theorem does apply.

There are, of course, more inference rules than just binary resolution. Even in this example, the proof uses demodulation. The theorem above can be extended to included additional rules of inference:

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*[2] *, 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] 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.

Remark. The theorem cannot be extended to apply to paramodulation from variables. An example is given below.

Proof. Conclusions reached by hyperresolution can also be reached by binary resolution, so that part of the theorem follows from the previous theorem. The results on factoring, paramodulation and demodulation follow from the fact that applying a substitution produced by unification preserves correct typings. The lemma that we need is that if p and r unify, then they have the same value type. If neither is a variable, this follows from the assumption that the axioms of T are correctly typed. (If one is a variable, this need not be the case.)

Suppose, for example, that $r = s$ is to be used as a demodulator on term t. The demodulator is applied by unifying r with a certain subterm p of t. Let σ be the substitution that performs this unification, so $p\sigma = r\sigma$. Then p and r, since they unify, have the same value type, and hence p, $p\sigma$, and $r\sigma$ all have the same value type. The type specification of equality must have the form $type(bool, = (X, X))$ for some type X; so r and s have the same value type, so $r\sigma$ and $s\sigma$ have the same value type. Hence $s\sigma$ and $p\sigma$ also have the same value type, and hence the result of replacing p in t by $s\sigma$ (the result of the demodulation) is a correctly typed term.

Example. This example will show that one cannot allow "overloading", or multiple type specifications for the same symbol, and still use implicit typing with guaranteed correctness. For example, 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, taken from Euclidean geometry, shows that the theorem cannot be extended to paramodulation from variables. In this example, *EQpt* stands for equality between points, *EQline* stands for equality between lines, $I(a, b)$ stands for point a incident to line b, and $p_1(u)$ and $p_2(u)$ are two distinct points on line u. The types here are boolean, point, and line. Axioms (1) and (2) are correctly typed:

$$EQpt(x, y)|I(x, line(x, y)). \tag{1}$$

$$EQline(line(p1(u), p2(u)), u). \tag{2}$$

Paramodulating from the first clause of (1) into (2), we unify x with $line(p_1(u), p_2(u))$, and thus derive

$$EQline(y, u)|I(line(p1(u), p2(u)), line(line(p1(u), p2(u)), y)). \tag{3}$$

This conclusion is incorrectly typed since y is a point and u is a line.

Example. This simpler 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.

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 will always be written explicitly, but we use both $lambda(x, t)$ and $\lambda x.\, t$.

Our main objective in this section is to define the lambda unification algorithm. As we define it here, this is a non-deterministic algorithm: it can return, in general, many different unifying substitutions for two given input terms. As implemented in Otter-lambda, it returns just one unifier, making some specific choice at each non-deterministic choice point. 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);
    if (τ = failure)
        return failure;
    σ = σ ∘ τ; }
return σ
```

Here the call to `unify` is a recursive call to the algorithm being defined.

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, re-

jecting any substitution that assigns a value depending on x.[3] 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 in steps 1 and 3. These steps are 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.[4] If x occurs in t, but no masking subterm occurs as a second argument of Ap, we pick the 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 the choice of S is determined by heuristics that prove useful in this case. In the future we hope to implement a version of lambda unification that returns multiple unifiers by trying different sets S in step 3. Our proofs in this paper apply to the full non-deterministic lambda unification, as well as to any deterministic versions, unless otherwise specified.

Example. Lambda unification can lead to untypable proofs, for example those needed to produce fixed points in lambda calculus. As an example, if we unify

[3] 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.

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

$Ap(x, y)$ 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:[5]

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

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.*

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 $type(B, Ap(i(A, B), A))$.

Remark. A discussion of the relationship, if any, between lambda unification and the higher-type unification algorithms already in the literature is beyond the scope of this paper. The algorithms apply to different systems and have different definitions. Similarly, the exact relationship between lambda logic and various sytems of higher-order logic, if there is any, is beyond the scope of this paper (or any paper of this length).

4 Implicit typing in lambda logic

If we consider the no-nilpotents example in lambda logic, we can state the axiom of mathematical induction in full generality, and Otter-lambda can use lambda unification to find the specific instance of induction that is required. (See the examples on the Otter-lambda website.) The proof, obtained without relativizing to unary predicates, is correctly typable. This is not an accident: there are theorems about implicit typing that guarantee it.

We first give an example to show that the situation is not as straightforward as in first-order logic. If we use the axioms of group theory in lambda logic, must we relativize them to a unary predicate $G(x)$? As we have seen above,

[5] 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.

that is not necessary when doing first-order inference. We could, for example, put in some axioms about natural numbers, and not relativize them to a unary predicate $N(x)$, and as long as our axioms are correctly typed, our proofs will be correctly typed too. There is, however, reason to worry about this when we move to lambda logic.

In lambda calculus, every term has a fixed point. That is, for every term F we can find a term q such that $Ap(F, q) = q$. Another form of the fixed point theorem says that for each term H, we can find a term f such that $Ap(f, x) = H(f, x)$. Applying this to the special case when $H(f, x) = c * Ap(f, x)$, where c is a constant and $*$ is the group multiplication, we get $Ap(f, x) = c * Ap(f, x)$. It follows from the axioms of group theory that c is the group identity. On the other hand, in lambda logic it is given as an axiom that there exist two distinct objects, say c and d, and since each of d and c must equal the group identity, this leads to a contradiction. Looked at model-theoretically, this means it is impossible, given a lambda model M, to define a binary operation on M and an identity element of M that make M into a group.

Referees of other papers about Otter-lambda have complained about this and similar examples. The referee's point about this example was that I ought to relativize the group axioms to a unary predicate G. The point of this paper is that there are good theoretical reasons why I do *not* need to do that.

First, let us consider how to type the relevant axioms. Writing G for the type of group elements, 1 for the group identity, and $i(G, G)$ for the type of maps from G to G, we would have the following type specifications:

$$type(G, 1).$$
$$type(G, *(G, G)).$$
$$type(G, Ap(i(G, G), G).$$
$$type(i(G, G), lambda(G, G)).$$

In general, of course, we want $type(i(X, Y), lambda(X, Y))$, but the special case shown is enough in this example. According to these type specifications, the axioms are correctly typable, and when Otter-λ produces a proof, the proof turns out to also be correctly typable. This is not an accident, as we will see.

In defining type specifications for lambda logic, the following technicality comes up: Normally in predicate logic we tacitly assume that different symbols are used for function symbols and predicate symbols. Thus $P(P(c))$ would not be considered a well-formed formula. In lambda logic we do wish to be able to define propositional functions, as well as functions whose values are other objects, so we allow *Ap* both as a predicate symbol and a function symbol. However, except for *Ap*, we follow the usual convention that predicate symbols and function symbols use distinct alphabets. This is the reason for clauses (4) and (5) in the following definition.

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*$(i(U,V), lambda(U,V))$,[6] *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.

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.

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 sσ and tσ are correctly typed, and S assigns the same type to s, t, and sσ.*

Example. Let s be $Ap(X,w)$ and t be $a+b$. We can unify s and t by the substitution σ given by $X := lambda(x, x+b)$. 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. It may be that 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.

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

[6] 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.

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 arg 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) 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.

The next theorem mentions *paramodulation* and *paramodulation from a variable*. Readers not already familiar with these terms will find them explained in the last paragraph of the proof below.

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 typable 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 typable with respect to S, provided paramodulation from or into variables are not allowed, and paramodulation into or from terms $Ap(X, w)$ with X a variable is not allowed, and demodulators similarly are not allowed to have variables or $Ap(X, w)$ terms on the left.*

Remark. The example after Theorem 3 shows that the second restriction on paramodulation is necessary: otherwise we could paramodulate from $x + 0 = x$ into $Ap(X, x)$ getting $X := lambda(x, x + 0)$, but here Ap has value type $Prop$, which is not the type of $x + 0$.

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 varaibles 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 typable. 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 $P[z := q\sigma]$. (This is the promised explanation of *paramodulation.*) 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. We have disallowed paramodulation from or into variables in the statement of the theorem; therefore t and r are not variables. Let us write $Type(t)$ for the value type of (any term) t. Because $t = q$ is correctly typed, we have $Type(t) = Type(q)$. If neither t nor q is an Ap term, then $Type(t\sigma) = Type(q\sigma)$, since they have the same functor. If one of them is an Ap term, then by hypothesis it is not of the form $Ap(X, w)$, with X a variable. 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 $Ap(X, w)$. 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. The fixed point argument which shows that the group axioms are contradictory in lambda logic requires 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 found by Otter-λ from a correctly typed input file. In particular, in lagrange3.in we have correctly typed axioms, so 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 typable 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. The fact that the theoretical result agrees with the results of running the program is a good thing.

Remarks. (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-lambda *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. Once Otter-λ finds a correctly typable proof, we can check by hand (and could easily check by machine) that it is correctly typable. Nevertheless it is of interest to be able to set a flag in the input file that enforces type-safe unification. In Otter-λ, if you put `set(types)` in the input file, then only certain lambda unifications will be performed, and those unifications will always be type-safe.

Spefically, *restricted* lambda unification means that, when selecting a masking subterm, only a second argument of Ap or a constant will be chosen. This is the restriction imposed by the flag `set(types)`. We now 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 typable 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.

6 Some examples covered by Theorem 5

It remains to substantiate the claims made in the abstract and introduction, that the theorems in this paper justify the use of implicit typing in Otter-λ for

the various examples mentioned. The first theorems apply in generality to any partial implementation of non-deterministic lambda unification, used in combination with resolution and paramodulation, but disallowing paramodulation into and from variables. Only Theorem 5 applies to Otter-lambda specifically, when the *set(types)* command is in the input file. We will now check explicitly that interesting examples are covered by this theorem.

Let us start with the "no nilpotents" example. It 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 example the type of *Ap* is the one needed for mathematical induction: $type(Prop, Ap(i(N, Prop), N))$, so the type of the second arg of *Ap* is N; but the axioms include a constant o for the zero of the ring. This is not a serious problem: we can simply replace o in the axioms by $zero(0)$, and give *zero* the type specification $type(R, zero(N))$. The term $zero(0)$ is not a constant, so it won't be selected as a masking term (where it would interfere with the proof of Theorem 5). But it will be treated essentially as a constant elsewhere in the inference process; and if we were worried about that, we could use a weight template to ensure that it has the same weight as a constant and hence will be treated *exactly* as a constant. On the logical side we have the following lemma to justify the claim:

Lemma 1. *Let T be a theory with at least one constant c. Let T^* be obtained from T by adding a new function symbol f, but no new axioms. Then (i) T^* plus the axioms $c = f(x)$ is conservative over T.*

(ii) If T contains another constant b and we let A^o be the result of replacing c by $f(b)$ in A, then T proves A if and only if T^* proves A^o.

(iii) There is an algorithm for transforming any proof of A^o in T^* to a proof of A in T.

Proof. (i) Every model of T can be expanded to a model of T^* plus $c = f(x)$ by interpreting f as the constant function whose value is the interpretation of c. The completeness theorem then yields the stated conservative extension result.

(ii) A^o is equivalent to A in T^* plus $c = f(x)$, so by (i), A^o is provable in T^* plus $c = f(x)$ if and only if T proves A. In particular, if T^* proves A^o then T proves A. Conversely, if T proves A and we just replace c with $f(b)$ in the proof, we get a proof of A^o in T^*.

(iii) The algorithm is fairly obvious: simply replace every term $f(t)$ in the proof with c. (Not just terms $f(b)$ but any term with functor f is replaced by c.) Terms that unify before this replacement will still unify after the replacement, so resolution proof steps will remain valid. The axioms of T^* plus $c = f(x)$ are converted to axioms of T plus $c = c$. Paramodulation steps remain paramodulation steps and demodulations remain demodulations. Since no variables are introduced, paramodulations that were not from or into variables are still not from or into variables. That completes the proof of the lemma.

This lemma shows us that logically, the formulation of the no-nilpotents problem with $zero(0)$ for the ring zero is equivalent to the original formulation with a constant o for the ring zero; and Theorem 5 directly applies to the formulation

with $o(0)$. In practice, if we run Otter-lambda with o replaced by $zero(0)$ in the input file, we find the same proof as before, but with o replaced by $zero(0)$. In essence this amounts to the observation that o was never used as a masking term in lambda unification in the original proof. Technically we should run the input file with $zero(0)$ first. Theorem 5 guarantees that if we find a proof, it will be well-typed. The lemma guarantees that we can convert it into a proof of the original formulation using a text editor to replace all terms with functor $zero$ by the original constant o.

We conclude with another example. Bernoulli's inequality is

$$(1 + nx) < (1 + x)^n \qquad \text{if } x > -1 \text{ and } n > 0 \text{ is an integer.}$$

Otter-lambda, in a version that calls on MathXpert [2] for "external simplification", is able to prove this inequality by induction on n, being given only the clausal form of Peano's induction axiom, with a variable for the induction predicate. The interest of the example in the present context is the fact that three types are involved: real numbers, positive integers, and propositions. The propositional functions all have N, the non-negative integers, for the ground type, but the types are not disjoint: N is a subset of the reals R. Moreover, the left-hand side of the inequality uses n in multiplication, so if multiplication is typed to take two real arguments, the inequality as it stands will not be well-typed.

The solution is to introduce a symbol for an injection map $i : N \to R$. The inequality then becomes

$$(i(1) + i(n)x) < (i(1) + x)^n$$

This formulation is well-typed, if we type i as a function from N to R. Again, in the definition of exponentiation we have to use 0 for the natural number zero, and $zero(0)$ for the real number zero, so that all the constants will have type N. If that is done, Theorem 5 applies, and we can be assured that the inference steps performed by Otter-lambda proper will lead from well-typed formulas to well-typed formulas. However, the theorem does not cover the external simplification steps performed by MathXpert. To ensure that these do not lead to mis-typed conclusions, we have to discard any results containing a minus sign or division sign, as that might lead out of the domain of integers. Problems involving embedded subtypes also arise even in typed theorem provers or proof checkers, so it is interesting that those problems are easily solved in lambda logic. The interested reader can find the input and output files for this and other examples on the Otter-lambda website.

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. MathXpert Calculus Assistant, software available from (and described at) www.HelpWithMath.com.
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.