

First Order Proof for Higher Order Logic Theorem Provers (abstract)

Joe Hurd

Computing Laboratory
University of Oxford,

goals [8], and a calculus with specific rules for equality is also important for this application.

11. L. C. Paulson. A generic tableau prover and its integration with Isabelle. *Journal of Universal Computer Science*, 5(3), March 1999.
12. Lawrence C. Paulson. Isabelle: A generic theorem prover. *Lecture Notes in Computer Science*

Implicit Typing in Lambda Logic

Michael Beeson¹

San José State University, San José, Calif.
beeson@cs.sj su. edu,
[www. cs. sj su. edu/facul ty/beeson](http://www.cs.sj su. edu/facul ty/beeson)

Abstract. Otter-lambda is a theorem-prover based on an untyped logic Otter, so it uses resolution proof search, supplemented by demodulation and paramodulation for equality reasoning, but it also uses a new algorithm. 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. If the axioms can be typed this way then the consequences should be typed this way as well. The main theorem of the paper shows that the ability to type by Otter-lambda, if type-safe lambda unification is used, and if demod-

II

of saying $x + 0 = 0$, we would say $R(x) \rightarrow x + 0 = 0$, or in clausal form, $\neg R(x) \mid x + 0 = 0$. (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

IV

Definition 1. *A type specification is an expression of the form $\text{type}(R, f(U, V))$, 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*

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 $Q \mid$

VI

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*

Example

VIII

even xy . In this paper, Ap will always be written explicitly, but we use both $\lambda(x, t)$ and $x \cdot t$.

XII

(4) *all type specifications with symbol lambda have the form
type(i*

(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 $\lambda x. Ap(r, q)$ and t is $\lambda x. Ap(R, Q)$ and lambda unification succeeds by unifying these terms as if they were first-order terms.

If the term r occurs in A , then r occurs in some $C[\mathcal{B}]$, so by the correctness of $T[\mathcal{B}]$, there is a type specification in S as required.

(iii) *each occurrence of each term r that occurs in A*

the form $Ap(X, w)$. Hence $Type(r) = Type(r') = Type(t) = Type(q)$, which is what had to be shown.

Now consider demodulation. In this case we have already deduced $t = q$ and $P[z := t]$ and we conclude $P[z := q]$, where the substitution is produced by lambda unification of t with some subterm of $P[z :=]$. Taking $r = t$, 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 $type(V, Ap(i(U, V)))$, 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- 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- 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- does find a proof using this input file, which is consistent with our argument above that fixed-point

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- and covers the examples posted on the Otter- website, some of which have been ment-24ed here. Otter- never uses paramodulation into or from variables, so that hypothesis of the above theorems is always satisfied. But Otter- 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- 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-

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 Leo is a Subsystem of Ω_{MEGA}

Leo has been realized as a part of the Ω_{MEGA} framework. This framework (and thus also Leo) is implemented in CLOS [25], an object-oriented extension of Lisp.

itself employ a Henkin

OMEGA: show-commands leo-interactive
[...]
DECOMPOSE: Applies


```

(implies (p (and (a m) (or (b m) (c m))))
  (p (and (a m) (dc-20735 c b))))))}
=====
Clause cl4 is #<Justified by ((CNF)) on (cl2)> :
cl4(1.5|1):{(+(p (and (a m) (or (b m) (c m)))))}
=====
Clause cl5 is #<Justified by
((RES 1 1) (RENAMING {(dc2 --> dc37)})) on (cl3 cl4)> :
cl5(1|2):{(-(= (p (and (a m) (dc37 c b)))
  (p (and (a m) (or (b m) (c m))))))}
=====
Clause cl7 is #<Justified by
((UNI {(?h113 --> [Iam ?h122 ?h123.m])
  (?h107 --> [Iam ?h120 ?h121.m])
  (?h101 --> [Iam ?h111 ?h112. (?h111 (?h113 ?h111 ?h112))])
  (?h100 --> [Iam ?h105 ?h106. (?h106 (?h107 ?h105 ?h106))])
  (dc37 --> [Iam ?h98 ?h99. (or (?h100 ?h98 ?h99)
    (?h101 ?h98 ?h99))])})
  (RENAMING {}}) on (cl5)> :
cl7(0|2):{NIL}
=====
Clause cl14 is #<Justified by ((CNF)) on (cl7)> :
cl14(0|3):{NIL}
=====
===== clauses in proof: 7 =====

```

OMEGA:

We now slightly modify our example problem and obtain a much harder one.

\exists y $r d ()$ $2(d ()$ u $)$ $2 2 d ($ $2 2 d (r d d ()$ $2(d ()$ $($ $1 1 11 1$ $)$ $(p p)(p)(p)(p)p$ n

THEORY-LIST (SYMBOL-LIST) Theories whose definitions will be expanded: [()]
Expanding the Definitions...
[...]

OMEGA: show-clauses
===== BEGIN =====
The set of

cladetails.

not employ optimizations and implementation tricks that are well known in the first-order community) and therefore the refutation of this clause set is not very

5. LE - is an Automated Theorem Prover for classical higher-order provers; see Sections 5 and 6. Below we initialize mega with the program `see` and then call `and` and `premid` and `auto` and `theorem` prover for classical higher-order logic. Within mega it can be applied. Leo is rst and

Leo's Architecture and Main Loop

Leo's basic architecture adapts the set of support approach. The four cornerstones of Leo's architecture (see Fig. 1) are:

⌈

⌋

Step 13 (Integrate to SOS) In the last step Leo integrates all pre-uni ed clauses in UNIFIED into the sorted store SOS. Forward and/or backward sub-
sumption is employed depending on ending




```
Start proving ...
Loop: (100sec left)
#1 (SOS 2 USABLE 0 EXT-QUEUE 0 F0-LIKE 4)
(99sec left)
#2 (SOS 1 USABLE 1 EXT-QUEUE 0 F0-LIKE 4)
(98sec left)
[...]
(96sec left)
#9 (SOS 3 USABLE 4 EXT-QUEUE 0 F0-LIKE 12)
[...]
Calling bliksem process 22267 with time resource 50sec .
PARSING BLIKSEM OUTPUT ...
Bliksem has found a saturation.
[...]
(96sec left)
#10 (SOS 10 USABLE 5 EXT-QUEUE 0 F0-LIKE 20)
[...]
(94sec left)
#21 (SOS 39 USABLE 9 EXT-QUEUE 0 F0-LIKE 60)
Calling bliksem process 22454 with time resource 50sec .
bliksem Time Resource in seconds:
PARSING BLIKSEM OUTPUT ...
Bliksem has found a proof.
Bliksem's time:
; cpu time (non-gc) 0 msec user, 0 msec system
; cpu time (gc)    0 msec user, 0 msec system
; cpu time (total) 0 msec user, 0 msec system
; real time 174 msec
; space allocation:
; 416 cons cells, 0 symbols, 15,720 other bytes,
```

Leo has initially been implemented as a demonstrator system for extensional higher-order resolution in the context of the author's PhD thesis [4]. The experiments carried out with Leo so far, in

SET171+3 $\forall X_0 ; Y_0 ; Z_0 : X (Y Z) = (X Y) (X Z)$
SET611+3 $\forall X_0 ; Y_0 : (X$

Assumptions:	$\forall B; C; x. [x \in (B \cup C) \rightarrow x \in B \vee x \in C]$	(1)
	$\forall B; C; x. [x \in (B \cap C) \rightarrow x \in B \wedge x \in C]$	(2)
	$\forall B; C [B = C \rightarrow B \subseteq C \wedge C \subseteq B]$	(3)
	$\forall B; C [B \subseteq C \rightarrow C \subseteq B]$	(4)
	$\forall B; C [B \subseteq C \rightarrow C \subseteq B]$	(5)
	$\forall B; C [B \subseteq C \rightarrow \forall x. x \in B \rightarrow x \in C]$	(6)
	$\forall B; C [B = C \rightarrow \forall x. x \in B \rightarrow x \in C]$	(7)
Proof Goal:	$\forall B; C; D [B \subseteq (C \cap D) \rightarrow (B \subseteq C) \wedge (B \subseteq D)]$	(8)

Table 3. SET171+3: The First-Order TPTP En/BPC 1 ID EI 04)

(1) $\forall B;C;D:B (C \supset D) = (B \supset C) \supset (B \supset D)$	clause initialization def.-expansion, cnf B; C; D Skolem const.
(2) $[(\exists x:Bx (Cx / Dx)) = (\exists x:(Bx / Cx) (Cx / Dx))]^F$	unification constraint
(3) $[(\exists x:Bx (Cx / Dx)) = ?(\exists x:(Bx / Cx) (Cx / Dx))]$	f-extensionality x new Skolem constant
(4) $[(Bx \supset (Cx / Dx)) = ?((Bx / Cx) \supset (Cx / Dx))]$	B-extensionality
(5) $[(Bx \supset (Cx / Dx)) \supset ((Bx / Cx) \supset (Cx / Dx))]^F$	cnf, factor., subsumption
(6) $[Bx]^F$	
(7) $[Bx]^T [Cx]^T$	propositional problem!
(8) $[Bx]^T [Dx]^T$	
(9) $[Cx]^F [Dx]^F$	propositional reasoning
(10)	

Table 4. Problem SET171+3: Solution in LEO

For the experiments with Leo and the cooperation of Leo with the first-order theorem prover Bliskem, λ -abstraction as well as the extensionality treatment inherent in Leo's calculus [4] is used. This enables a theorem prover Henkin-complete proof system for set theory. In the above example SET171+3, Leo generates the application of functional extensionality to push unification constraints down to base terms.

Some problems are immediately mapped by recursive definition expansion (without extensionality reasoning) and normalisation into the empty clause such that proof search does not even start; an example is SET646+3. Some problems require several rounds of extensionality processing within Leo's set-of-support based proof search procedure before the bag of first-order like clauses turns into a refutable

ing systems such as automated theorem provers, computer algebra systems, or model generators into the architecture.

3.1 Cooperation via multiple inference rules

The use of the external systems is modelled by inference

3.2 Cooperation via a single inference rule

In order to overcome the problem

layer of the ants blackboard architecture. This is not an ad hoc solution, but

{ When the above suggestion of a successful joint proof attempt is

type variables during the search for a proof. A third alternative is the most realistic. We provide precisely the relevant axioms and previously proven theorems, with the correct type variable instantiations, and try to prove C follows. In general, of course, we cannot know which of the A are relevant. However, for the Grundlagen theorems, we can extract this information from the Automath proof terms. Using this information, we obtain thousands of theorems of the form

$$[A'_i \wedge \dots \wedge A'_i$$

Co-Synthesis of New Complex Selection
Algorithms and their Human Comprehensible
XML Documentation

Theorem, which was originally proven in 1976 by Appel and Haken who used computer programs to check a very large number of cases.

However, for a mathematician it is unsatisfying to know that there exists a solution or no solution for a problem, because thousands, hundreds of thousands or millions of states have been explored by a theorem prover whose rules have been verified, and at the same time, not be able to comprehend the tedious machine-generated proofs and not be able to draw conclusions from the automated proof.

context-specific rules as how to decompose the proof graph, when to generate

Fig. 1. Scenario for Knowledge-based Synthesis

or relationships between them have been defined. Figure 2 lists the major types of the SEAMLESS framework.

Basic types *Bool, Integer, Constants, Vars, String*
Object logic *Atom, Clause, Algebraic Expression, Constraint*
Synthesis types *Precondition, Postcondition, Program, Proof*

Fig. 2. Types for Program Synthesis

- to facilitate the automatic synthesis of human comprehensible XML-based documents;
-

The advantage of using graphterms for encoding higher-order formulas is

can be processed by the t

6 Experimental Results

A major advantage of combining higher-order program synthesis and document synthesis is that the generated proof terms describe programs at the

sis or program documentation and merely exploit the correspondences between XML data and logical terms, e.g., [BDHG,LR03,SR01]. The presented framework distinguishes itself from related approaches (cf. [Pec04,GKP96,Oks05]) in which computer have been used for the complexity analysis of unsolved mathematical problems. The results synthesised by SEAMLESS are constructive, comprehensible and can be manually checked for correctness, provided that the sizes of the

Fig. 10. Poset P

lemma 2 *Let P be a poset as visualised in Figure 11. The 3-rd largest element of P is computed by at most 8 comparisons.*

Fig. 11. Poset P

Proof. **algorithm 3**

1. Compare a_5 and a_7 .
selecting the 2-nd largest element takes at most $f(2, 10.0) = 7$ comparisons.
- (b) $a_5 < a_7$.

Mixing Finite Success and Finite Failure in an Automated Prover

Alwen Tiu¹, Gopalan Nadathur², and Dale Miller³

¹ INRIA Lorraine/LORIA

² Digital Technology Center and Dept of CS, University of Minnesota

³ INRIA & LIX, Ecole Polytechnique

Abstract. The operational semantics and typing judgements of modern programming and specification languages are often defined using relations and proof systems. In simple settings, logic programming languages can be used to provide rather direct and natural interpreters

programming system, not in a purely logical way at least, even though they

$\Sigma \sigma$

definitions is an extension of work by Schroeder-Heister [SH93], Eriksson [Eri91],

the following classes of formulas:

Level 0: $G := \neg \mid A \mid G \mid G \mid G \mid G \mid \exists x:G \mid \forall x:G$
 Level 1: $D := \neg \mid \neg \mid A \mid D \mid D \mid D \mid D \mid \exists x:D \mid \forall x:D \mid \exists x:D \mid G \mid \supset D$
 atomic: $A := p \mid t_1 \mid \dots \mid t_n$

Here, atomic formulas A in level 0 j6264 Tf 19.9052 962648

Concrete syntax The concrete syntax for Level 0/1 prover follows the syntax of

more subtle. Consider the case where both eigenvariables and logic variables are present in a negative goal, for example, consider proving the goal

$\neg x:zy:($

z : p

in : $n \quad (n \quad p) \quad p$

```
onep (in X M) (dn X) M.Q           % bound input
one  (out X Y P) (up X Y) P.       % free output
one  (taup P) tau P. tau           % tau
one  (match X X P) A Q :=
```

We now define a notion of equivalence between processes, called bisimulation. It is formally defined as follows: a relation \approx is a bisimulation, if it is a symmetric relation such that for every $(P; Q) \in \approx$,

1. if $P \xrightarrow{x(z)} P^0$ and $(P; Q) \in \approx$ is a free action, then there is Q^0 such that $Q \xrightarrow{x(z)} Q^0$
2. if $Q \xrightarrow{x(z)} Q^0$ and $(P; Q) \in \approx$ then there is P^0 such that $P \xrightarrow{x(z)} P^0$ for every


```
sat P top.
sat P (and A B) := sat P A, sat P :.
sat P (or A B) := sat P A; sat P :.
sat P (boxMatch X Y A) := (X = Y) => sat P A.
sat P (diaMatch X Y A) := (X = Y), sat P A.
sat P (boxAct X A) := pi P1\ one P X P1 => sat P1 A.
sat P (diaAct X A) := sigma P1\ one P X P1, sat P1 A.
sat P (boxOut X A) := pi Q\ onep P (up X) Q => nabl a y\ sat (Q y)
```

an unevaluated expression, and its evaluation is

9 F

$\frac{\Sigma ; \sigma \triangleright \quad \cdot \quad \sigma \triangleright}{\Sigma ; \sigma \triangleright} \text{init} \quad \Sigma ; \cdot \quad \mathcal{B} \quad \Sigma ; \mathcal{B} \quad \cdot \quad \mathcal{C}$



[MPW93] Robin Milner, Joachim

T_{PS}: A Theorem Proving System for Church's Type Theory

Chad E. Brown

Universität des Saarlandes, Saarbrücken, Germany, cebrown@ags.uni-sb.de

T_{PS} [1, 2] is a theorem proving system providing support for automated

System Description: The Metis Proof Tactic

Joe Hurd

Computing Laboratory
University of Oxford,
joe.hurd@comlab.ox.ac.uk

