Theorem proving for untyped constructive λ-calculus: implementation and application

Allan Ramsay, *Department of Computation, UMIST, PO Box 88, Manchester M60 1QD, UK. E-mail: Allan.Ramsay@umist.ac.uk*

Abstract

This paper presents a theorem prover for a highly intensional logic, namely a constructive version of property theory [25] (this language essentially provides a combination of constructive first-order logic and the λ-calculus). The paper presents the basic theorem prover, which is a higher-order extension of Manthey and Bry's model generation theorem prover for first-order logic [14]; considers issues relating to the compile-time optimisations that are often used with first-order theorem provers; and shows how the resulting system can be used in a natural language understanding system.

Keywords: higher order logic, model generation

1 A constructive intensional logic

For various reasons, the idea of a language which allows you to construct abstractions and apply them to terms, as in the λ-calculus, and to combine these operations with the truth functional connectives of predicate logic, is extremely tempting. Section 4 will show some applications of such a higher-order logic for natural language processing, and there are plenty of others. It is well known, however, that simply adding the λ-calculus and predicate logic together opens the way to the paradoxes of negative self-reference – the Liar, Russell's set, and so on.

The classical way out of this is to place restrictions on what can be said, either by imposing a hierarchy of abstraction levels and banning propositions of one level from commenting on the truth of propositions at the same or a higher level [28, 24]; or by demanding that abstraction is used only for producing subsets of sets which are themselves well-founded (constructible), as in Zermelo-Fraenkel set theory [9]. Turner approaches the matter by allowing you to say whatever you want, but then placing constraints on what can be proved [25].

Turner (personal communication) argues that shifting the constraints from what can be *said* to what can be *proved* allows you to deal with various cases which seem perfectly reasonable, but which would be blocked by some of the syntactic restrictions, e.g. it is not possible to say *'John knows he knows everything that Peter knows, and Peter knows he knows everything that John knows'* within some such approaches. Placing syntactic restrictions on the arguments of intensional operators also turns out to be awkward at various points in natural language processing: section 4 provides a number of examples which show the advantages of using a language which allows untyped abstraction in this domain.

Turner's analysis involves taking a classical treatment of first-order logic and adding

 ©Oxford University Press

λ-abstraction and β-reduction to it (or at any rate, operations which look extremely like λ-abstraction and β-reduction). In order to avoid the paradoxes, however, he constrains the circumstances under which you are allowed to perform λ-abstraction. The constraints he chooses are enough to make the underlying logic consistent (in other words, to avoid the paradoxes), and makes the paradoxes UNSTABLE [8]. [3] showed that it is impossible to provide a normal form for Turner's version of the logic, and hence very difficult to implement a theorem prover for it. The current paper takes a constructive treatment of first-order logic, allows unrestricted use of both λ-abstraction and β-reduction, but avoids the paradoxes by placing constraints on the assumptions that can be used in a well-founded proof.

The logic which we will use, which we will call $\Lambda(C)$ for constructive λ-calculus, extends first-order logic as follows:

$\Lambda(C)$-1 If A is a formula of first-order logic then it is a formula of $\Lambda(C)$, and if t is a term of first-order logic then it is a term of $\Lambda(C)$.

$\Lambda(C)$-2 If A is a formula of $\Lambda(C)$, possibly including free occurrences of x, then $\lambda x A$ is a term of $\Lambda(C)$.

$\Lambda(C)$-3 If t and t' are terms of $\Lambda(C)$ then $t.t'$ is a formula.

The proof theory for this logic is obtained by adding the rules in Fig. 1 to a standard set of natural deduction rules for constructive logic (see e.g. [26], or any text on constructive logic).

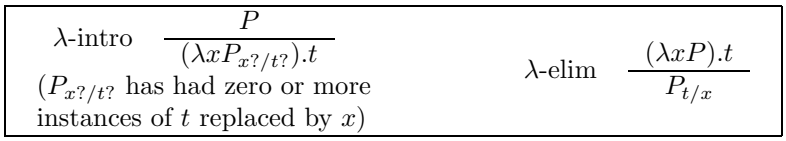

FIG. 1. Natural deduction rules for $\Lambda(C)$

Theorem 1.1 shows that you can add λ-intro and λ-elim to a standard set of natural deduction rules without introducing proofs of $\perp$.

Theorem 1.1 Soundness of $\Lambda(C)$

If there is no proof of $\perp$ from α using the standard rules of constructive logic then any proof of $\perp$ from α using λ-intro and λ-elim introduces some formula containing an irreducible redex[1].

PROOF. Suppose (a) that there is a proof P_α of $\perp$ from α using λ-intro and λ-elim in addition to the standard rules, where no member of α contains an irreducible redex; (b) that there is no proof of $\perp$ from α without λ-intro and λ-elim; and (c) there is no set α' for which (a) and (b) also hold, where the corresponding proof $P_{\alpha'}$ contains fewer applications of λ-intro and λ-elim.

Consider the first use in this proof of $\perp$ from α of either (i) λ-elim to change some formula $(\lambda x A).t$ into $A_{t/x}$, or (ii) λ-intro to change some formula $A_{t/x}$ into $(\lambda x A).t$

[1] A redex $(\lambda x A).t$ is irreducible if there is no sequence of applications of λ-elim which will produce a term with no occurrences at all of $(\lambda y B).s$. Irreducible redexes will only occur if we use an untyped version of λ-abstraction, which we do here for the reasons given in Section 1 and Section 4.

(there must be one, since otherwise α would have supported a proof of $\perp$ from the standard rules alone). In case (i) we can obtain a proof of $\perp$ from $\alpha \cup A_{t/x}$, and in case (ii) we can obtain one from $\alpha \cup (\lambda x A).t$, each of which omits the relevant step, and hence involves fewer applications of these rules, contradicting assumption (c) (note that $(\lambda x A).t$ is irreducible iff $A_{t/x}$ is, so that the first step which adds either of these to α will not introduce a formula containing an irreducible redex unless there was already one there). ∎

The point of this theorem is that any proof of $\perp$ from a set which is consistent under the first-order rules must introduce some formula containing an irreducible redex (since otherwise every subproof would satisfy the conditions of the theorem). If we take the constructive view of λ-applications as promissory notes for proofs, or for programs that would produce proofs [26], then formulae containing irreducible redexes are promises that can never be fulfilled.

Caveat emptor: Theorem 1.1 shows that the logic is sound so long as you avoid formulae with irreducible applications of λ-terms. It does *not* prevent you from saying things which would have been better left unsaid. Consider the proofs in Fig. 2, where R stands for the term $\lambda x(\neg x.x)$:

(1)	$R.R \vdash R.R$	(ASS)
(2)	$R.R \vdash \neg(R.R)$	(λ-elim applied to (1))
(2′)	$R.R \vdash (R.R) \rightarrow \perp$	($\neg$-elim applied to (2))
(3)	$R.R \vdash \perp$	($\rightarrow$-elim applied to (2′, 3))
(4)	$\vdash R.R \rightarrow \perp$	($\rightarrow$-intro applied to (3))
(4′)	$\vdash \neg R.R$	($\neg$-intro applied to (4)

(1)	$\neg R.R \vdash \neg R.R$	(ASS)
(2)	$\neg R.R \vdash \neg\neg(R.R)$	(λ-elim applied to (1))
(2′)	$\neg R.R \vdash (\neg R.R) \rightarrow \perp$	($\neg$-elim applied to (2))
(3)	$\neg R.R \vdash \perp$	($\rightarrow$-elim applied to (2′, 3))
(4)	$\vdash \neg R.R \rightarrow \perp$	($\rightarrow$-intro applied to (3))
(4′)	$\vdash \neg\neg R.R$	($\neg$-intro applied to (4)

FIG. 2. Russell's set is strange

It seems that the claim that Russell's set is a member of itself is provably false, but so is the claim that it is not a member of itself, and between them these lead to a contradiction. Note, however, that $(\lambda x(\neg x.x)).(\lambda x(\neg x.x))$ *cannot* be fully reduced.

Note that there are a number of proof steps that will introduce formulae containing irreducible redexes, but which will not allow you to infer $\perp$ from A unless A was already inconsistent by the standard first-order rules. You can infer A *or* $R.R$ from A, for instance, and you can infer P from $\lambda x P.(R.R)$ if x does *not* occur free in P. Theorem 1.1 bans such proofs. We don't lose much by banning them: any theorem which does not itself contain an irreducible redex and which could proved using them could be proved without them. We can, in particular, allow λ-intro to produce vacuous λ-abstractions (which could lead to the introduction of a formula of the form $\lambda x P.(R.R)$) without fear that we will thereby introduce proofs of $\perp$.

2 Satchmo for $\Lambda(C)$

The theorem prover we will present for $\Lambda(C)$ is developed by extending [14]'s first-order theorem prover Satchmo. The original presentation of Satchmo is very effective for puzzles (where all the information that is present is required for solving the problem, but it is hard to see how to use it), but can perform very poorly on problems where a lot of the information that is present is irrelevant. [17] and [11] show how to avoid some of the pathological behaviour of the basic Satchmo algorithm in such circumstances: unfortunately these techniques, like most other optimisations for first-order theorem provers [10, 2], rely on a static analysis of the problem. Optimisations that rely on static analysis of the initial problem statement do not work for intensional logics. Section 3 shows how to recover such optimisations dynamically.

The original presentation of Satchmo is unsuitable for our purposes, since it assumes a classical version of predicate logic, so that you can prove P by showing that $\neg P$ is unsatisfiable, and you can also use equivalences such as $((P \to Q) \to R) \leftrightarrow ((Q \to R)\&(P \textit{ or } R))$ which are not available in constructive logic. We therefore need to adapt it so that it does work properly for constructive logic.

We do this in two stages: first we have to convert our problem into an appropriate normal form, and then we have to adapt the basic Satchmo engine to work constructively with this normal form.

Normal form:

The construction of a normal form proceeds in three stages.

(i) We start by making very straightforward textual changes, to make standard logical form look a bit more like Prolog and to get rid of existential quantifiers.

NF-1 Replace `(A & B)` by `(A', B')` and `(A or B)` by `(A'; B')`, where `A'` and `B'` are the normal forms of `A` and `B`.

NF-2 Replace `not(A)` by `(A' => absurd)`

NF-3 Replace `P => (Q => R)` by `((P & Q) => R)'`.

NF-4 Skolemise away existential quantifiers, and remove all universal quantifiers.

(ii) Separate the result of (i) into Horn and non-Horn clauses, and convert the Horn clauses to ordinary Prolog.

PL-1 If the normal form of `P` is atomic then assert it as a Prolog fact.

PL-2 If the normal form of `P` is `(Q, R)` then deal with `Q` and `R` individually.

PL-3 If the normal form of `P` is `(Q;R)` then assert `split(Q; R)` as a Prolog fact.

PL-4 If the normal form of `P` is `(K => Q)` where `Q` is atomic then assert `Q :- K` as a Prolog rule.

PL-5 If the normal form of `P` is `(K => (Q, R))` then deal with `(K => Q)` and `(K => R)` individually.

PL-6 If the normal form of `P` is `(K => (Q; R))` then assert `split(Q; R) :- K` as a Prolog rule.

(iii) Perform any optimisations that you can on these. We will return to this in Section 3.

Constructive Satchmo

Once we have the problem converted to normal form, we can use the following adaptation of the basic model generation algorithm.

MG-1 If you can prove `A` by using Prolog facts and rules then you can prove it.

MG-2 You can prove `A` if you can prove `split(P, Q)` and any of (i) `P => A` and `Q => A`, or (ii) `P => A` and `not(Q)`, or (iii) `not(P)` and `Q => A` [2].

MG-3 In order to prove `(A => B)`, you have to add `A` to your set of Prolog facts and rules and then show that you can prove `B`. This corresponds to $\rightarrow$-introduction.

(MG-1) and (MG-2) are exactly as in the original presentation of Satchmo, except that since Satchmo works by trying to show that the hypotheses + the negation of the goal are unsatisfiable it always tries to prove `absurd`, whereas a constructive version has to show that the goal itself is provable from the hypotheses (though the second pair of routes through (MG-2) allow you to do this slightly indirectly). (MG-3) is introduced because Satchmo relies on the classical equivalence between $((P \rightarrow Q) \rightarrow R)$ and $(Q \rightarrow R)\&(P \; or \; R)$ when constructing normal forms. This equivalence is no longer available: if we want to prove $P \rightarrow Q$ we have to use $\rightarrow$-introduction. In particular, if you want to prove `not(P)`, as you might as a consequence of using (MG-2(ii)) or (MG-2(iii)), then you will have to do this by using (MG-3) to prove `P => absurd`. Fig. 3 provides a skeletal implementation of this.

```
% You can prove A either directly
prove(A) :-
    A.

% or by proving (P or Q), (P => A) and (Q => A)
prove(A) :-
    split(P; Q),
    \+ (P; Q), % check you haven't tried this already
    prove(P => (A or absurd)),
    prove(Q => (A or absurd)).

% To prove (P => A), assert P and try to prove A
% (with some funny bookkeeping to tidy up after yourself)
(P => A) :-
    assert(P),
    (prove(A) -> retract(P); (retract(P), fail)).
```

FIG. 3. Basic constructive Satchmo

The key non-cosmetic differences between this and Satchmo are that (i) this version implements a constructive version of first-order logic rather than a classical one, and (ii) it is slightly more direct when faced with clauses of the form $((P \rightarrow Q) \rightarrow R)$. Most of the work in Satchmo is performed in the backward chaining phase where the Prolog facts and rules are being used to prove specific goals. By converting $((P \rightarrow Q) \rightarrow R)$ to `R :- (P => Q)`, we ensure that this rule is activated when it is

[2] (MG-2) supports proofs of formulae such as $P(a)$ *or* $P(b) \rightarrow \exists x P(x)$ for which there are no UNIFORM proofs [15, 12], so that Satchmo does not fit their criteria as the basis of a logic programming *language*. It's still a perfectly useful starting point for a theorem prover.

required, at the cost of having to prove `P => Q` by asserting `P` and showing that `Q` follows from it. If we convert $((P \to Q) \to R)$ to `R :- Q` and `split(R; P)`, we end up having to explore the consequences of asserting `P` anyway.

2.1 Adding abstraction and λ-reduction

According to λ-intro and λ-elim, $(\lambda xA).t$ and $A_{t/x}$ are equivalent, and according to Theorem 1.1 there is no problem with this so long as none of your assumptions or hypotheses contain irreducible instances of $(\lambda xP).t$. We would therefore like to add the following step to the normal forming process:

NF- Replace `lambda(X, P):T` by $\texttt{P}_{\texttt{T/X}}$ [3].

This would eliminate all instances of $(\lambda xA).t$ before we ever started trying to use the underlying inference engine, so that including such expressions in our problem statement would have no effect whatsoever on the performance of the theorem prover. Unfortunately, the definition of $\Lambda(C)$ also allows formulae of the form $x.y$, where x and y are variables. If it did not, then the language would not really be all that different from ordinary first-order logic. But since it does, we have a problem with producing the correct normal form for such cases. We need one final normal form rule:

PL-7 If the normal form of `P` is `Q => (X:A)`, where `X` is a variable, then replace it by `split(X:A) :- Q`.

We also have to extend the inference engine to take account of these new cases, as shown in Fig. 4. The new clause for `prove(A)` reflects the decision that clauses with λ-applications involving uninstantiated functions as their heads should be used forwards, like clauses with disjunctive conclusions. The point here is that since we do not know what the conclusion of such a clause is, we have no way of telling whether it is likely to be useful. We therefore leave them out of the backwards chaining part of the proof procedure, and simply allow them to emerge when there is nothing more obvious to try.

The clause for proving `(P:T)` says that if you know what `P` is then you should actually work with the β-reduced version. This is guaranteed to work precisely because if `(P:T)` were provable from the original problem statement then we will have either replaced it directly by $\texttt{P}_{\texttt{T/X}}$ during the normal forming process, or we will eventually do so when we explore the consequences of `split(P:T)`.

The program outlined above provides a sound and reasonably efficient theorem prover for $\Lambda(C)$. The soundness is guaranteed by Theorem 1.1 and Theorem 2.1. The reasonable efficiency is inherited from Satchmo, together with the fact that by proving $(\lambda xA).t$ from $A_{t/x}$ we start working backwards as soon as we possibly can. Completeness is any case unavailable: $\Lambda(C)$ provides a mapping between formulae and terms which enables you to axiomatise various logics *within* the theory. In particular you can provide an encoding of the integers and a statement of Peano arithmetic; it is therefore clear that the set of valid formulae, which includes all formulae of the form

[3] I write `lambda(X, P):T` as the Prolog equivalent of $\lambda X(P).T$ because Prolog already uses the full stop as its end of clause marker.

```
% you can also prove A if (lambda(X, P):T) => A
% and you can prove P(T/X)
prove(A) :-
    split(P:T),
    nonvar(P),
    fully_reduce(P:T, R),
    (R => A).

% To prove (P:T), try proving P(T/X) instead
(P:T) :-
    nonvar(P),
    fully_reduce(P:T, R),
    prove(R).
```

FIG. 4. Extending Satchmo to cover λ-intro and λ-elim.

$\mathcal{P} \rightarrow A$ where $\mathcal{P}$ is the axiomatisation of Peano arithmetic and A is some arithmetical truth, is not recursively enumerable, whereas the theorems of $\Lambda(C)$ are[4].

Theorem 2.1 The algorithm outlined in Fig. 3 and Fig. 4 is sound.

PROOF. Suppose that the algorithm is not in fact sound, i.e. that there is a proof of `absurd` from some set of clauses {`A1, ..., An`} using the algorithm in Fig. 3 and Fig. 4 which would not have led to such a proof just using the algorithm in Fig. 3. At some point the proof must have either (i) used a splitting rule to derive $\mathtt{P}_{\mathtt{T/X}}$ from `lambda(X, P):T`, or (ii) proved `lambda(X, P):T` by proving $\mathtt{P}_{\mathtt{T/X}}$. In case (i) we could have proved `absurd` from {`A1, ..., An, lambda(X, P):T`}, and in case (ii) we could have proved it from {`A1, ..., An,` $\mathtt{P}_{\mathtt{T/X}}$}. In either case we have a proof of `absurd` using one less application of the relevant rule. We can repeat this until there are no applications of either of the rules from Fig. 4, in which case we have a set {`A1, ..., An, K1, ..., Km`} which supports a proof of `absurd` just using the rules in Fig. 3. ∎

This algorithm provides the basis for implementing a theorem prover for $\Lambda(C)$. Just as the inclusion of (MG-2) meant that neither the classical nor the constructive versions of first-order Satchmo provide a good basis for a logic programming language, the decision to allow (a) untyped λ-abstractions and (b) non-rigid intensional operators in the *consequents* of sequents means that this extended algorithm is thoroughly unsuitable as the basis for a higher-order logic programming language. The contrast between our goals and the decisions underlying λProlog [16] are captured in the statement that "[...] a definite clause is to be viewed as the definition of a procedure whose name is given by the head of A. A consideration of this fact indicates that predicate variables are not *usefully* permitted in the head position in A" [op.cit., p.508, my italics]. $\Lambda(C)$ is aimed explicitly at tasks where the natural axiomatisation does seem to require predicate variable in heads, and if that makes the computation harder then so be it.

[4] $\Lambda(C)$ also permits an internal description of a number of other logics for which the sets of valid formulae are not r.e., notably [22]'s default logic.

3 Optimisations

The program described above is 'reasonably efficient' – as efficient, that is, as something based on Satchmo could be expected to be. As noted earlier, however, Satchmo is very effective for some kinds of problem, less so for others. There are known optimisations for Satchmo, particularly for lessening the impact of irrelevant disjunctive clauses. We will consider a number of these in the context of the extension of Satchmo to deal with $\Lambda(C)$.

3.1 Deleting and reinstating pure literals

The first move is to introduce Kowalski's notion of 'pure literals' [10]. Suppose that we have a clause of the form `p(X, Y) :- p1(X, Y), ..., pk(X, Y), ..., pn(X, Y)`, but there is no clause with `pk(U, V)` as its head. Then clearly there is no point in using this rule when attempting to prove `p(a, b)`, since the k-th subgoal is bound to fail. So we may as well remove it from our clause set. The subgoal `pk(U, V)` is said to be PURE.

But this might be the only clause which supports proofs of `p(U, V)`. In that case, removing it may well make it possible to delete some other clause, which may ... Kowalski shows how this kind of 'gangrene' can lead to quite dramatic reductions in the problem statement. The effect tends not to be quite so dramatic in the context we are working in (meaning postulates for lexical semantics and discourse relations in natural language), but there are in any case two problems with it.

(i) Kowalski's original presentation marked a literal L as being pure if there was no clause containing a complementary literal L' which would unify with it. We, however, are working with equality as well as intensionality. Consider the set of clauses in Fig. 5.

$male(f)$
$male(f)\&parent(f,a) \rightarrow father(f,a)$
$parent(j,a)$
$f = j$

FIG. 5. Rule set with an apparently pure literal

It seems as though there is nothing which could support $parent(f,a)$, since the only potentially relevant literal, $parent(j,a)$, does not unify with it. This suggests that we can delete the rule for proving $father(f,a)$. This is clearly too strong, since the presence of the equality means that we *can* prove $parent(f,a)$, so that we should not delete this rule. We are therefore restricted to saying that we can delete a rule if it contains a subgoal `g(t1, t2)` for which there is no clause whose head has `g` as its functor and arity 2. This is weaker than Kowalski's notion, and hence is less likely to lead to drastic reductions in the search space.

(ii) To make matters worse, however, the fact that we have clauses with completely underspecified heads means that we actually have no idea at all what literals might actually be provable. We therefore cannot simply throw away clauses with pure literals, since it is entirely possible that a literal may become *impure* as a result of the forward application of some clause resulting from a formula with an intensional

consequent. The best we can do is to put them to one side until they become impure, and then reinstate them. Thus instead of deleting a clause whose antecedent contains pure literals, we store it in such a way that it can easily be reinstated. Fig. 6 shows the form in which `father(f, a) :- male(f), parent(f, a)` would be stored if `parent(f, a)` was pure, and we include an extra step in the treatment of conditional proofs, as in Fig. 7.

```
impure(father(X, Y),
       (father(f, a) :- male(f), parent(f, a))).
```

FIG. 6. Storing a clause with a pure literal for later reinstatement

```
impurify(G) :-    % failure driven to get all cases
    impure(G, (A :- B)),
    assert((A :- B)),
    impurify(A),  % recursion reverses 'gangrene' effect
    fail.
impurify(_).

prove(A => B) :-
    assert(A),
    impurify(A, CLAUSE),
    (prove(B) ->
        retract(A);
        (retract(A), fail)).
```

FIG. 7. What to do when a literal becomes impure (bookkeeping omitted)

Split rules transform themselves into conditional proofs, since in each case the result of a split rule is to introduce a request for a proof that the new information would lead to a proof. Fig. 3 dealt with such requests by adding the antecedent of the clause to the set of facts and then attempting to show that the goal is provable under these new circumstances. Fig. 7 simply makes sure that anything that the new facts would help with is made available before the proof continues.

This introduces many of the benefits of pure literal deletion in a context where clauses containing pure literals may suddenly become available as a result of moves which could not have been predicted. The cost is a small fixed time search for reinstatable clauses every time you undertake a conditional proof.

3.2 Relevance

Satchmo can be made to perform very poorly if you include disjunctive clauses where one or both of the disjuncts is actually irrelevant. The problem is that each use of such a clause, say $A \rightarrow (B \text{ or } C)$, introduces two conditional proofs, namely $B \rightarrow G$ and $C \rightarrow G$, where G is the top-level goal you are actually trying to prove. If B is not going to contribute to a proof of G then $B \rightarrow G$ will be provable precisely if G itself is, and likewise for $C \rightarrow G$. But then at least one of the branches introduced by using $A \rightarrow B \text{ or } C$ is at least as hard as just proving G without using this clause.

[17] and [11] show that you can deal with this problem by banning the use of a `split`

clause until it has been shown that its consequents *will* contribute to a proof of `G`. The current implementation continues to take the approach of dynamically reinstating such a clause when at least one of its disjuncts has shown up in a failed attempted proof. The key is that instead of including such a clause in the database directly, we assert a rule which will itself add the clause when its consequents have been shown to be relevant. If we had, for instance, a clause like `split(p(X); q(X)) :- g1(X), ..., gn(X)` we would replace it by the clauses shown in Fig. 8.

```
p(_Y) :-
    \+ clause((split(p(X); q(X)) :- g1(X), ..., gn(X))),
    assert((split(p(X); q(X)) :- g1(X), ..., gn(X))),
    fail.
q(_Y) :-
    \+ clause((split(p(X); q(X)) :- g1(X), ..., gn(X))),
    assert((split(p(X); q(X)) :- g1(X), ..., gn(X))),
    fail.
```

FIG. 8. Allow `split(p(X); q(X))` if either of them is relevant

The first of these says that if you find yourself trying to prove `p(t)`, and you don't already have this `split` clause available to you, then add it to the database, and likewise for `q(t)`. This provides a very cheap way of implementing the requirement that at least one of the disjuncts should be potentially relevant to something that you actually want to prove. It is less easy to provide a cheap test to ensure that both disjuncts are desirable – you have to choose between a cheap test that may still allow some undesirable cases through [17], and a more expensive one which is more rigorous [11].

Note that the implementation in Fig. 8 dissociates the instantiation of the goal you are attempting to prove from the rule which gets reinstated. Doing this means that we do not add multiple versions of the same rule, at the cost of adding ones that are potentially still irrelevant. It is easy enough to implement the more precise version, simply by replacing the anonymous variable `_Y` in the head of the clause by the specific variable `X` that appears in the rule to be reinstated.

3.3 Keyhole optimisations

The optimisations in Sections 3.1 and 3.2 improve the performance of the theorem prover by cutting out parts of the search space which need not be explored. The fact that the problem statement has been turned largely into executable Prolog code suggests that there are other possible optimisations that may arise from thinking about the kinds of optimisation that can be applied to Prolog programs. We include two specific optimisations that are applied to individual clauses without regard to their place in the overall proof space. If we can make the Prolog versions of clauses run more quickly we will get a uniform increase in speed, which is not as significant as the improvements that result from global pruning but is still worth having.

Loop-checking

It is very easy to write Prolog programs that get into infinite loops. Of course if you are actually writing programs, it is really your responsibility to ensure that this does not happen. The programs we are concerned with here, however, are generated automatically from sets of statements in $\Lambda(C)$, and it is not appropriate to restrict what can be said in this language just because of what the theorem prover might do. Fig. 9, for instance, shows the rule we use to capture the notion that intervals are partially ordered:

$$\forall I_1 : \{interval(I_1)\}$$
$$\forall I_2 : \{interval(I_2)\}$$
$$\forall I_3 : \{interval(I_3)\}(I_1 \leq I_2 \& I_2 \leq I_3 \rightarrow I_1 \leq I_3)))$$

FIG. 9. Partial order of intervals

In order to stop rules like this producing infinite loops, we thread a sequence of 'labels' [6] through the Prolog form of the rule. Labels carry arbitrary information about the progress of a proof, and can be used for a variety of purposes. For loop checking, we include an abbreviated copy of the proof stack in the label. We ensure that each rule that *could* lead to a loop adds an entry to this proof stack, and that each such rule also includes a check to see whether we are currently in a loop, so that the Prolog corresponding to the clause in Fig. 9 looks as in Fig. 10.

```
le(LABEL0/LABEL5, I1, I3) :-
    checkloop(le(I1, I3),  LABEL0),
    interval(I1, LABEL0/LABEL1),
    interval(I2, LABEL1/LABEL2),
    interval(I3, LABEL2/LABEL3),
    le([le(I1, I3) | LABEL3]/LABEL4, I1, I2),
    le([le(I1, I3) | LABEL4]/LABEL5, I2, I3).
```

FIG. 10. Using labels for loop-checking

The labels are threaded through the clause, so that information can be passed from one subgoal to the next. For loop checking we add the current goal to the goal stack at the points where we are about to call something which might lead us into a loop, and we start this call by explicitly looking to see whether we are in a loop. Clearly the definition of `checkloop` is critical here – if you define it too tightly you will lose proofs which are actually available, if you define it too loosely you will still get into loops. For practical purposes we choose to define it quite tightly, so that we have some confidence that the algorithm will terminate (it is therefore, of course bound to be incomplete. That's a choice you have to make – the more loops you cut out, the more potentially provable theorems you miss).

Labels have a variety of other purposes: we also use them for instance, for carrying around the equivalence classes that result from using rules relating to equality, which we use for dynamically rewriting clauses (rewriting the entire clause set, as proposed

by [7], is very expensive if you have large clause sets: we prefer to rewrite clauses as we use them, using information encoded in the label).

Ground goals

The final optimisation relates to the proof of ground goals. It often happens that at a given point in a proof your current goal is ground, but there are multiple ways of proving it. This is particularly significant in situations where it is possible to prove that two items are equal, since you may have one set of proofs relating to `a` and another relating to `b`: when you realise that `a` and `b` are the same thing, you have to be careful about how you use these two sets of proofs.

We exploit the fact that our rules are converted to Prolog, so that we can use any programming tricks we care to from Prolog, to lessen the impact of this problem. If the head of a rule is ground at the point when it is called, we know that there is no point in finding multiple proofs of it. We therefore place a Prolog cut, to be invoked iff the head was ground at the point when the clause was called. This changes the clause for $\leq$ given in Fig. 10 to the one in Fig. 11.

```
le(LABEL0/LABEL5, I1, I3) :-
    % Is the head ground?
    (ground([I1, I2]) -> GROUND = true; GROUND = fail),
    checkloop(le(I1, I3),  LABEL0),
    interval(I1, LABEL0/LABEL1),
    interval(I2, LABEL1/LABEL2),
    interval(I3, LABEL2/LABEL3),
    le([le(I1, I3) | LABEL3]/LABEL4, I1, I2),
    le([le(I1, I3) | LABEL4]/LABEL5, I2, I3),
    % Was the head ground? If so, commit yourself to this proof
    (GROUND -> !; true).
```

FIG. 11. Ground clauses only have one proof

4 Application in natural language processing

We use the theorem prover described above for various tasks in natural language processing. The effectiveness of a theorem prover depends at least partly on the nature of the tasks for which it is to be used. The system described here is designed to work in a context where ordinary first-order theorem proving dominates, but a certain amount of higher-order/intensional inference also has to be carried out. The following sections illustrate the kinds of place where intensional operators may be required in NLP: the aim is not to defend our linguistic treatments (see [18, 19, 21] for further discussion) so much as to show how they make use of the inference engine.

4.1 Referring expressions

In order to construct referring expressions, you have to find descriptions which say enough about the item being discussed to pick it out, without saying more than is strictly needed – over-precise referring expressions disrupt the 'coherence' of the text

containing them. [1] uses a mixture of λ-calculus and first-order logic in deriving appropriate descriptions, without apparently worrying about the foundational issues. [4, 5] discuss ways of ensuring that such expressions are near-minimal, and explain why this matters. Generating an expression of the form *'the man'*, for instance, in order to refer to some individual j requires you to show that $\lambda X man(X)$ satisfies some property like the following, where $\Delta[S]$ is the speaker's view of the current discourse state [20, 23]:

$$\lambda P((\Delta[S] \rightarrow P.j)\&\forall Y((\Delta[S] \rightarrow P.Y) \rightarrow Y = j)) \qquad (Def.\ descr.)$$

This expresses a variant on Russell's treatment of definite NPs as making claims about the existence of some unique object satisfying the given description. *Def. descr* says that the speaker's view of the current discourse state (which should be the same as the hearer's view of it, but may not be) supports a proof that j satisfies the description; and there is no other item for which the same information supports such a proof – in other words, we are not saying that there is only one such object, we are saying that there is only one thing for which the common ground supports a *proof* that it is of the specified kind.

General reasoning about expressions such as *Def. descr* requires the expressive power of $\Lambda(C)$. The theorem prover outlined in this paper cannot be used for generating appropriate descriptions, but it can be used for understanding them. In particular, [20] and [23] show how to use this treatment when faced with definite NPs which 'refer' to items which have not been mentioned, but whose existence is *implied* by the context (such as the NPs *'the loser'* and *'the winner'* in *'John and Mary are going to have a race. The loser will have to buy the winner a drink.'*).

4.2 Lexical meaning postulates

Natural language includes the words *'true'* and *'false'*, and it also includes a number of other words whose detailed meaning involves commentary on the truth or falsity of other expressions. The presence of *'true'* and *'false'* themselves clearly opens the way to the paradoxes:

(*) (*) I am now lying.
(*) What I am now saying is false.

More interestingly, various modifiers such as *'not'* and *'only'* comment on the truth of both the current sentence and some other proposition.

(*) (*) I didn't steal your *bike*.
(*) I didn't steal *your* bike.
(*) I didn't *steal* your bike.

(*) (*) I only borrowed your *bike*.
(*) I only borrowed *your* bike.
(*) I only *borrowed* your bike.

Each sentence in (4) carries the basic message that *'I stole your bike'* is false, but they also carry some extra implication – that I stole something else of yours, or that I stole someone else's bike, or that I did something else to your bike. Similarly each sentence in (8) says that the sentence *'I borrowed your bike'* is true, but this time they say that something else is false – that I didn't borrow your car, or that I didn't borrow Martin's bike, or that I didn't steal your bike.

Spelling out the meanings of words like *'not'* and *'only'* requires you to produce untyped intensional meaning postulates – untyped because these words can apply to phrases of (almost) any syntactic type, so that their semantic analysis must apply to phrases of almost any semantic type: intensional because they comment on the truth or falsity of other parts of the sentence. We need something like

$$\forall P \forall X(only(P,X) \rightarrow (P.X \& \exists P'(similar(P,P') \& \neg P'.X))) \qquad (Ax.\text{'only'})$$

This axiom says that if *only(P, X)* holds, where P is some property and X is an arbitrary entity (possibly a property itself), then P does hold of X, but there is some property P' which is similar to P, and which you might have expected to hold of X, but which in fact fails to do so. How you measure similarity of properties, and how you determine that some property was implicated by the context, are questions which are beyond the scope of the current paper. What matters is that without using an untyped intensional logic of the kind investigated here, you cannot even *ask* the relevant questions, let alone answer them.

Such axioms can, again, only be manipulated if you have a theorem prover for something like $\Lambda(C)$ (similar remarks apply to non-factive adjectives like *'fake'* and *'alleged'* which comment on the truth of the modified expression – if I bought a fake Picasso then the one thing you can be sure of is that I did not buy a Picasso). It is worth emphasising that the axiom for *'only'* involves quantification over properties of unrestricted type. We don't know whether the stressed argument of *'only'* will be a determiner, or an adjective, or a noun, or a verb, or a VP, or ... We therefore cannot afford to place restrictions on its type in the axiom.

4.3 Discourse relations

Large discourses are structured by, among other things, 'rhetorical relations' between utterances. Rhetorical structure theory (RST) [13], for instance, develops characterisations of a large number of relations such as consequence, elaboration, concession, ... Any attempt to axiomatise such relations must involve quantification over propositions, using axioms such as

$$\forall P \forall Q(concession(P,Q) \rightarrow (P \& Q \& default(P \rightarrow \neg Q)))) \qquad (Ax.concession)$$

This says that if the discourse relation of concession holds between P and Q then the speaker is committed to the truth of both P and Q, but acknowledges that P would normally provide evidence for $\neg Q$. Some axiom like this is required for understanding sentences like

(*) John loved his work, even though he was extremely poorly paid.

(12) says that John loved his work, and that he was extremely poorly paid, and admits that the second of these might lead you to expect the first not to be true. Just as above, reasoning with axioms like *Ax.concession* requires a logic which provides predicates that represent truth and falsity: we are currently conducting experiments with such axioms, using the theorem prover sketched above to investigate their consequences. The use of defaults in this kind of axiom is problematic, but at least, as

noted in Section 2.1, $\Lambda(C)$ provides the expressive power for encoding the various notions of proof *within* the logic itself[5], so that we can explore the various options.

5 Conclusions and further work

The theorem prover outlined above is in regular use with an NLP system which deploys meaning postulates of the kind sketched in Section 4, with approximately 450 rules, of which 8 have disjunctive consequents, 20 have intensional consequents, and 17 lead to equalities.

With these meaning postulates, we can perform the inferences required for assimilating an utterance in around a second for simple sentences – obviously the more complex a sentence is the longer it will take to process it, but even very simple sentences have rather substantial sets of consequences in this framework.

It is difficult to get a feel for how efficient an inference engine is unless you know how hard the problems it is dealing with are, and that depends on the particular properties of the axiom set – not just how many disjunctive sequents, how many intensional ones, and how many involving equality there are, but also how they interact. It is, however, interesting to compare the effects of the various optimisations. Consider the task of constructing a model which would make the following sentence true:

(*) John fired his secretary.

The task we are undertaking here involves showing that this is consistent, by *failing* to derive a proof of $\bot$ from (13) itself and all our background knowledge, and then reading a (partial) model off this failed proof. The formal paraphrase of (13) is given in Fig. 12.

$$
\begin{array}{l}
claim(\exists B :\{B\ is\ interval\ \ \&\ \ ends\_before(ref(\lambda C(speech\_time(C,1))),B)\} \\
\qquad \exists D :\{aspect(simple,B,D)\} \\
\qquad\quad \forall A :\{memb(A,D)\} \\
\qquad\qquad \theta(A,agent,ref(\lambda E(named(E,John)))) \\
\qquad\qquad\ \ \&\ fire(A) \\
\qquad\qquad\ \ \&\ A\ is\ event \\
\qquad\qquad\ \ \&\ \theta(A, \\
\qquad\qquad\qquad object, \\
\qquad\qquad\qquad ref(\lambda F secretary(F) \\
\qquad\qquad\qquad\qquad \&\ owns(ref(\lambda G(salient(G,H)\ \&\ m(G))),F))))
\end{array}
$$

FIG. 12. Formal paraphrase of *'John fired his secretary'*

The model construction process involves tasks such as anchoring the referring expressions, disambiguating which sense of *'fire'* is meant (terminate someone's contract/let off an explosive charge in order to expel a projectile), deciding what to do with the utterance, investigating its ramifications (e.g. noting that if you fire someone then you must previously have employed them). This mixture of activities involves reasoning with a mixture of intensional meaning postulates (e.g. whatever someone

[5] though the inference engine described here requires considerable fine-tuning if it is to be effective for such embedded logics.

claims should be treated as being true) and ordinary everyday facts (if you fire someone then you must have been employing them, the act of firing them terminates their contract, you can only employ human beings, ...). Dealing with referring expressions also requires intensional reasoning, since you have to either find something that satisfies the given property or 'accommodate' the presupposition that there is such an entity [27]: accommodated referring expressions introduce new information by the back door – you could easily utter (13) in a situation where your hearer knew who John was, but did not know he had a secretary. Under those circumstances you would expect them to introduce someone who was John's secretary into their model of the world. (13) involves a simple referring expression (*'John'*) which can be dereferenced with respect to background knowledge, a pronoun (*'his'*) which has to be anchored to John, and a complex reference (*'his secretary'*) which has to be accommodated; it also involves the discourse relation of claiming, a lexical item (*'fire'*) which has to be disambiguated by using background knowledge, and a fair amount of general background knowledge. Constructing the model in Fig. 13 is therefore a fairly challenging task.

$owns(\#174, \#196)$	$fire2(\#197)$
	$\theta(\#197, agent, \#174)$
$\#184 > \#195$	$\theta(\#197, object, \#196)$
	$type(\#197, event)$
$end(\#195, \#198)$	$termination(\#197, \#199)$
$completion(\#195, \#198)$	
$type(\#195, instant)$	$employ(\#199)$
	$\theta(\#199, object, \#196)$
$secretary(\#196)$	$\theta(\#199, agent, \#174)$
$type(\#196, human)$	
$accommodated(secretary(\#196))$	$aspect(simple, \#198, \#197)$
$accommodated(owns(\#174, \#196))$	

FIG. 13. Model for *'John fired his secretary'*

The table in Fig. 14 shows the effects of the various optimisations discussed in Section 3. It turns out that pure literal deletion and relevance checking interact in unexpected ways[6]. Strong purification corresponds to deleting clauses if they contain literals for which there is no relevant Horn clause, weak purification corresponds to deleting them if there is no relevant clause at all. If you use strong purification with the relevance check from Section 3.2 then clauses will be deleted because their only support comes from some disjunctive clause; but disjunctive clauses are now only triggered if they would be helpful. The problem is that the clauses that the disjunctive clauses would have been helped by (which are their triggers) will have been purified, and hence will never be accessed.

The two columns marked ± groundedness show the effect of blocking repeated proofs of the same ground fact, as discussed in Section 3.3.

[6] Well I didn't expect it, and it took a lot of tracking down!

	+groundedness	-groundedness
Unoptimised	1.62	1.98
Strong purification	0.38	0.42
Relevance checking	1.55	2.04
Weak purification	0.54	0.59
Relevance + weak pur.	0.52	0.59

FIG. 14. Relative effects of optimisations (times in seconds)

The key observation is that the optimisations do improve the performance. It's what optimisations are supposed to do, of course, but it's always reassuring when they do. It is striking, however, that the effect of relevance checking with this particular problem is extremely marginal. If we simply add relevance checking to the basic system, we get a small improvement, if we add it to the weak form of purification we get a small improvement, but the best performance comes from the strong version of purification, which *cannot* be combined with relevance checking. It seems likely that the relative effectiveness of different combinations will depend on the exact mix of sequents in the problem statement. Relevance checking is not all that useful in the current context because, rather simply, there is not much irrelevant information in the meaning postulates. The use of strict purification, where clauses are only available for use if there is a *Horn* clause that could potentially help with their derivation, is made legitimate by the fact that we reinstate purified clauses when the forward chaining part of the algorithm introduces the required information. The move to a dynamic version of pure literal deletion was forced on us by the fact that we are working in an intensional context, where it is not possible to permanently delete clauses, since they may be impurified at any time. It turns out to work very nicely with Satchmo, since it means that we can be much more ruthless about what we delete.

The most notable remaining problem is that the performance drops off rather sharply as long sequences of NL utterances are processed and assimilated. This arises because the basic logical form of a sentence remains available as the basis for further inferences throughout a discourse. The difficulty is to stop it being *re*-used for inferences that have already been performed as well. This problem is currently under investigation.

References

[1] D Appelt. Planning English referring expressions. *Artificial Intelligence*, 26:1–33, 1985.

[2] W Bibel, R Letz, and J Schumann. Bottom-up enhancements of deductive systems. Technical report, Technische Universitat München, 1987.

[3] M Cryan and A M Ramsay. A normal form for Property Theory. In *Proceedings of the 14th International Conference on Automated Deduction (CADE-14)*, volume 1249 of *Lecture Notes in Artificial Intelligence*, pages 237–251, Berlin, 1997. Springer-Verlag.

[4] R Dale. *Generating Referring Expressions in a Domain of Objects and Processes*. MIT Press, Cambridge, Mass., 1992.

[5] R Dale and E Reiter. Computational interpretations of the Gricean maxims in the generation of referring expressions. *Cognitive Science*, 19(2):233–263, 1995.

[6] D M Gabbay. Labelled deductive systems. Technical report, Dept. of Computing, Imperial College, 1989.

[7] J Gallier, P Narendran, D Plaisted, S Raatz, and W Snyder. An algorithm for finding canonical sets of ground rewrite rules in polynomial time. *Journal of the ACM*, 40(1):1–16, January 1993.

[8] H Herzberger. Notes on naive semantics. *Journal of Philosophical Logic*, 11:61–102, 1982.

[9] T J Jech. *Lectures in Set Theory, with Particular Emphasis on the Method of Forcing*. Springer Verlag (Lecture Notes in Mathematics 217), Berlin, 1971.

[10] R Kowalski. A proof procedure using connection graphs. *JACM*, 22(4):572–595, 1975.

[11] D W Loveland. Near-horn Prolog and beyond. *Journal of Automated Reasoning*, 7:1–26, 1991.

[12] D W Loveland and G Nadathur. Proof procedures for logic programming. In D M Gabbay, C J Hogger, and J A Robinson, editors, *Handbook of logic in artificial intelligence and logic programming: volume 5, logic programming*, pages 163–234. Clarendon Press, Oxford, 1998.

[13] W C Mann and S A Thompson. Rhetorical structure theory: Toward a functional theory of text organization. *Text*, 8(3):243–281, 1988.

[14] R Manthey and F Bry. Satchmo: a theorem prover in Prolog. In *Proceedings of the 9th International Conference on Automated Deduction (CADE-9)*, volume 310 of *Lecture Notes in Artificial Intelligence*, pages 415–434, Berlin, 1988. Springer-Verlag.

[15] D Miller, G Nadathur, F Pfennig, and A Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[16] G Nadathur and D Miller. Higher-order logic programming. In D M Gabbay, C J Hogger, and J A Robinson, editors, *Handbook of logic in artificial intelligence and logic programming: volume 5, logic programming*, pages 499–590. Clarendon Press, Oxford, 1998.

[17] A M Ramsay. Generating relevant models. *Journal of Automated Reasoning*, 7:359–368, 1991.

[18] A M Ramsay. Focus on "Only", and "Not". In Y Wilks, editor, *Proceedings of the 15th International Conference on Computational Linguistics (COLING-94)*, pages 881–885, Kyoto, 1994.

[19] A M Ramsay. Aspect and aktionsart: fighting or cooperating? In J-I Tsujii, editor, *Proceedings of the 16th International Conference on Computational Linguistics (COLING-96)*, pages 889–894, Copenhagen, 1996.

[20] A M Ramsay. Does it make any sense? updating = consistency checking. In K Turner, editor, *The Semantics-Pragmatics Interface from Different Points of View*, London and Amsterdam, 1999. Elsevier.

[21] A M Ramsay. Dynamic and underspecified semantics without dynamic and underspecified logic. In H Bunt, L Kievit, R Muskens, and M Verlinden, editors, *Computing Meaning*, volume 1, pages 208–220, Dordrecht, 1999. Kluwer Academic Publishers (SLAP 73).

[22] R Reiter. A logic for default reasoning. *Artificial Intelligence*, 13(1):81–132, 1980.

[23] H Seville and A M Ramsay. Making sense of reference to the unfamiliar. In M Kay, editor, *Proceedings of the 18th International Conference on Computational Linguistics (COLING-2000)*, pages 775–781, Universität des Saarlandes, July 2000.

[24] R.H. Thomason, editor. *Formal Philosophy: Selected Papers of Richard Montague*. Yale University Press, New Haven, 1974.

[25] R Turner. A theory of properties. *Journal of Symbolic Logic*, 52(2):455–472, 1987.

[26] R Turner. *Constructive Foundations for Functional Languages*. McGraw Hill, London, 1991.

[27] R van der Sandt. Presupposition projection as anaphora resolution. *Journal of Semantics*, 9:333–377, 1992.

[28] A N Whitehead and B Russell. *Principia Mathematica*. Cambridge University Press, Cambridge, 1925.

Received 19 June 2000. Revised 10 October 2000