# Backward Chaining

Roger Bishop Jones

**Abstract**

This document provides facilities for automatic reasoning based on backward chaining. They are intended to be similar in capability to refutation proof procedures such as resolution or semantic tableau, but in order to fit in better with interactive proof in ProofPower are not refutation oriented. The main target is a backchaining facility which searches for a proof of the conclusion of the current goal from premises and rules drawn from the assumptions and elsewhere.

Created: 2004/10/03

Last Modified Date: 2006/12/11 17:30:48

http://www.rbjones.com/rbjpub/pp/doc/t012.pdf

Id: t012.doc,v 1.5 2006/12/11 17:30:48 rbj01 Exp

# Contents

# 1 Introduction

SML

```
open_theory "basic_hol";
set_pc "basic_hol";
```

The facilities provided here are intended to work with a rule set held in a unifying term net and to provide efficient elaboration of chains of inference constructed by composing applications of rules made to match by unification. The rules would typically be generalised implications, but equivalences are equally acceptable.

The interface is intended to be consistent with matching the premises of a rule with the conclusions of other rules at the time when the rule base is established or augmented, or with unification on demand combined with cacheing, or by pure unification on demand.

The design of the data structures involved is therefore required to permit the rule base software to anticipate what unifications will be required with minimal prejudice to the search strategy or other aspects of the implementation of the chaining inference facility.

# 2 Unifying Backchaining

The idea here is to get an improvement in proof development productivity by getting a more of the necessary instantiation of intermediate results done automatically.

This is approached as a "backchaining" problem for two reasons:

1. the conclusion of the goal provides valuable information for constraining the search space
2. the assumption that a theorems will be used from left to right also helps avoid explosion of the search space

In fact, the approach we adopt here makes the search into the transformation of a single conjecture which initially grows in size as the problem is picked apart, but eventually collapses if a proof is successful.

## 2.1 Rule Management

The methods used for storing an retrieving rules have some significance for performance and effectiveness, though these are less important than the search strategy. For example, to maximise performance in rule matching a unifying rule database would be possible, which efficiently retrieves all the rules which match a target term.

The purpose of this signature is to make a clean interface to the rule end of the backchaining so that an implementation of backchaining can start out rather crudely in this area and perhaps be improved later.

SML

```
signature RuleManager = sig
```

**Description** This is the signature of types and functions for managing and using a collection of rules for chaining. At present oriented to back-chaining.

```
SML
|type RULEDB
```

```
SML
|end; (* of signature RuleManager *)
```

## 2.2 The Unifying Backchain Rule

```
SML
|signature UnifyingBackchaining = sig
```

SML

```
val unify_backchain_rule : (TYPE list) -> (TERM list)
    ->(THM * TERM list * TYPE list)
    ->(TERM * TERM list * TYPE list)
    ->     THM * ((TYPE * TYPE) list * (TERM * TERM) list) *
        ((TYPE * TYPE) list * (TERM * TERM) list);
```

**Description** This function is a variant on $term\_unify$ q.v. in which the first term to be unified is suppled as the right hand side of an implication which is the conclusion of some theorem. In addition to the substitutions necessary to unify the two terms, the function returns the left had side of the implication after performing on it the unifying substitution for the right hand side, and a function which will infer the instantiated right hand side as a theorem from a theorem whose conclusion is the instantiated left hand side.

Thus from a theorem:

$$thm = asms \vdash lhs \Rightarrow rhs$$

and a term $tm$:

*unify_backchain_rule avtyl avtml* (*thm*, *tml1*, *tyl1*) (*tm*, *tml2*, *tyl2*)

yields $(thm2, (thmtytyl, thmtmtml), (tmtytyl, tmtmtml))$, where:

**avtyl** is a list of type variables to be avoided

**avtml** is a list of term variables to be avoided

**thm** is a theorem whose conclusion is an implication the right hand side of which is to be unified with $tm$

**tml1** is the list of term variables which may be instantiated in $thm$

**tyl1** is the list of type variables which may be instantiated in $thm$

**tm** is a term to be unified with the right hand side of the implication in the conclusion of $thm$

**tml2** is the list of term variables which may be instantiated in $tm$

**tyl2** is the list of type variables which may be instantiated in $tm$

**thm2** is the instance of $tm$ resulting from application of the unifying substitution

**thmtytyl** is the list of pairs of types to be substituted in $thm$

**thmtmtml** is the list of pairs of term to be substituted in $thm$

**tmtytyl** is the list of pairs of types to be substituted in $tm$

**tmtmtml** is the list of pairs of term to be substituted in $tm$

SML

```
type BCONV;
```

**Description** This is the type of functions which play the role for backchaining analogous to that played by *CONV* in rewriting, i.e. something which takes a *TERM* and returns a theorem having that term as its conclusion.

From this description one might expect that the type *CONV* would suffice for this application, since the only difference is in the kind of theorem required, which would be an implication rather than an equation. However two further complications are introduced.

The first is that instantiation of variables in the target term is permitted, but not necessarily all of them. So the function must be given the list of variables which may be instantiated.

Secondly, we allow for the possibility that there may be considerable effort, which we may wish to avoid until the proof search in progress has been completed. So the function returns results in two stages. The first is the term on the left of the probable implication, the second is the implication as a theorem.

Definition

```
type BCONV = (TERM * TERM list)
    -> (TERM * TERM list * TERM * TERM list * unit -> THM);
```

SML

```
end; (* signature UnifyingBackchaining *)
```

SML

```
structure UnifyingBackchaining: UnifyingBackchaining = struct
```

...

SML

```
end;
open UnifyingBackchaining;
```

SML

```
datatype BCP =
    BcDone
|   BcFailed
|   BcIncomplete;

datatype BCS =
    BcLeft
|   BcRight;

(* In a Back Chain Tree, think of 'a as the type of goals
    and 'b as that of proofs *)

open CrudeTermNet;

datatype ('a, 'b) BCT =
```

```
    Bc∧ of BCP * ('a * ('a, 'b) BCT * ('a, 'b) BCT)
|   Bc∨ of BCP * ('a * BCS * ('a, 'b) BCT * ('a, 'b) BCT)
|   BcRules of BCP * ('a * ('b * ('a, 'b) BCT) UTMNET)
|   BcLeaf of BCP * 'a;

fun bct2bcp (Bc∧ (x,y)) = x
|   bct2bcp (Bc∨ (x,y)) = x
|   bct2bcp (BcRules (x,y)) = x
|   bct2bcp (BcLeaf (x,y)) = x;

fun bcp_and (BcDone, BcDone) = BcDone
|   bcp_and (BcFailed, _)    = BcFailed
|   bcp_and (_, BcFailed)    = BcFailed
|   bcp_and (_, _)           = BcIncomplete;

fun bcp_or (BcDone, _)          = (BcDone, BcLeft)
|   bcp_or (_, BcDone)          = (BcDone, BcRight)
|   bcp_or (BcFailed, BcFailed) = (BcFailed, BcLeft)
|   bcp_or (_, _)               = (BcIncomplete, BcLeft);

val bcp_or2 = fst o bcp_or;

fun utm_fold_or ((utm, bct), bcp) = (bcp_or2 (bcp, (bct2bcp bct)));
fun bct_utmnet_map bctm f bct = bctm f bct;

fun
  bct_map f (Bc∧ (bcp, (a, bct1, bct2))) =
    let val bct1' = bct_map f bct1;
        val bct2' = bct_map f bct2;
    in Bc∧ (bcp_and (bct2bcp bct1', bct2bcp bct2'), (a, bct1', bct2'))
    end
| bct_map f (Bc∨ (bcp, (a, bcs, bct1, bct2))) =
    let val bct1' = bct_map f bct1;
        val bct2' = bct_map f bct2;
        val (bcp', bcs') = bcp_or (bct2bcp bct1', bct2bcp bct2');
    in Bc∨ (bcp', (a, bcs', bct1', bct2'))
    end
| bct_map f (BcLeaf (bcp, a)) = f (BcLeaf (bcp, a))
| bct_map f (BcRules (bcp, (a, utmn))) =
    let val utmn' = utmnet_map utmn (fn (x,(y,z)) => bct_map f z);
        val bcp' = utmnet_fold utm_fold_or utmn' BcFailed
    in BcRules (bcp', (a, utmn))
    end;

fun bct_∧ (g as (c, (tm, tyl, tml))) =
```

```
    let val (lhs, rhs) = dest_∧ tm
        val lg = (c, (lhs, tyl, tml))
        val rg = (c, (rhs, tyl, tml))
    in Bc∧ (BcIncomplete, (g, BcLeaf (BcIncomplete, lg), BcLeaf (BcIncomplete, rg)))
    end;

fun bct_∨ (g as (c, (tm, tyl, tml))) =
    let val (lhs, rhs) = dest_∨ tm
        val lg = (c, (lhs, tyl, tml))
        val rg = (c, (rhs, tyl, tml))
    in Bc∨ (BcIncomplete, (g, BcLeft, BcLeaf (BcIncomplete, lg), BcLeaf (BcIncomplete, rg)))
    end;

fun bct_⇒ bct_rule_can (g as (c, (tm, tyl, tml))) =
    let val (lhs, rhs) = dest_⇒ tm
        val lg = (c, (lhs, tyl, tml))
        val rg = (c, (rhs, tyl, tml))
    in Bc∨ (BcIncomplete, (g, BcLeft, BcLeaf (BcIncomplete, lg), BcLeaf (BcIncomplete, rg)))
    end;
```

# 3 Rule Manager

SML

```
signature BcRuleManager = sig
```

**Description** The facilities provided here are intended to work with a rule set held in a unifying term net and to provide efficient elaboration of chains of inference constructed by composing applications of rules made to match by unification.

The interface is intended to be consistent with matching the premises of a rule with the conclusions of other rules at the time when the rule base is established or augmented, or with unification on demand combined with cacheing, or by pure unification on demand.

The design of the data structures involved is therefore required to permit the rule base software to anticipate what unifications will be required with minimal prejudice to the search strategy or other aspects of the implementation of the chaining inference facility.

I did think of trying to make the rule base independent of whether forward or backward chaining was envisaged, but I decided against.

SML

```
type RULEBASE;
```

**Description** This is the type of a collection of rules.

The parameters are:

SML

```
end; (* of signature BcRuleManager *)
```

# 4 INDEX