Computación y Sistemas Vol. 2 Nos. 2-3 pp.126-139
© 1999, CIC - IPN. *ISSN 1405-5546* Impreso en México

Using Difference Reduction for Generic Proof Search

Santiago Negrete
Departamento de Computación
ITESM-Morelos
Paseo de la Reforma 182-A
Col. Lomas de Cuernavaca
Cuernavaca, Morelos 62050
santiago@campus.mor.itesm.mx

Article received on September 2, 1998; accepted on February 22, 1999

Abstract

In this paper a new approach to generic theorem proving is introduced. We present a set of techniques to guide proof search in framework theories that works with different object theories encoded. The techniques are based on the principle of Difference Reduction *and programmed in a* Proof Plans *environment. We use presentations of logics in natural deduction style to test our techniques. Two example theorem proofs are included to show how the whole setting works.*

Keywords:

Framework theories, proof search, rewrite rules, unification, natural deduction.

1 Introduction

Logic has proven to be of prime importance in many areas of Science and, in particular, in AI and Computer Science. Many logics have been developed to contend with the various kinds of reasoning required by a vast number of research areas.

Automatic theorem proving techniques for these logics are often developed in an ad-hoc way for particular theories in which specific problems are representable. *Framework theories* (Harper *et al.*, 1992; Constable *et al.*, 1986; Coquand and Huet, 1988), have been proposed as meta-mathematical theories in which other theories may be represented and reasoned with uniformly. Hence, automating proof search in a framework theory gives the possibility of abstracting the proof process and make it applicable to a larger number of logics.

In this work, we introduce an approach to proof search in the Edinburgh Logical Framework (Harper *et al.*, 1992) that is not hard-wired to a particular object logic. The design is based on guiding proof search through constrained rewriting. The rewriting technique is called *Rippling* (Bundy *et al.*, 1993) and it has been previously applied to the domain of inductive proofs.

Rippling is implemented within the framework of Proof Plans (Bundy *et al.*, 1991). This framework consists of building proof plans for theorems from abstract specifications of proof techniques called *Methods*. Methods correspond to tactics programmed in a Proof Editor where the theorem can be represented and where a proof plan can be realized to construct the actual proof for the theorem.

Our work is focussed on logic presentations. These are logics encoded in framework theories. We use logics in natural deduction style because their presentations share a common structure and are particularly suitable to extract rewrite rules from their inference rules.

We have extended the Rippling technique in various ways to handle this more general case. A feature of our

approach is that, since the rewrite rules used in Rippling are extracted from the logic presentation in the framework theory, the proof mechanisms are independent of the logic at hand.

We present an example proof where all our techniques are used. Examples in other logics and a more technical exposition of the techniques described here can be found in (Negrete, 1996).

2 The Edinburgh Logical Framework

The *Edinburgh Logical Framework* (LF) (Harper *et al.*, 1992) is a formal system specifically designed as a framework theory. It is a typed lambda calculus with dependent function types which capture the notion of inference rule in a natural way. Object theories are represented in LF by specifying *signatures*. These are sets of typed identifiers which define term and formula constructors of the theory (e.g. connectives, operations, constant terms, etc.) and constants whose types represent the inference rules of the object theory. It is particularly useful to represent logics in natural deduction and Hilbert styles (Avron *et al.*, 1987). In our particular case, we call logic presentations those signatures that represent logics. In Figure1 there is an LF signature for intuitionistic propositional logic:

$$
\begin{array}{rcl}
o & : & Type \\
true & : & o \rightarrow Type \\
\supset, \wedge, \vee & : & o \rightarrow o \rightarrow o \\
\bot & : & o \\
\supset_i & : & \Pi_{A,B:o}(true(A) \rightarrow true(B)) \rightarrow true(A \supset B) \\
\supset_e & : & \Pi_{A,B:o} true(A \supset B) \rightarrow true(A) \rightarrow true(B) \\
\wedge_i & : & \Pi_{A,B:o} true(A) \rightarrow true(B) \rightarrow true(A \wedge B) \\
\wedge_{el} & : & \Pi_{A,B:o} true(A \wedge B) \rightarrow true(A) \\
\wedge_{er} & : & \Pi_{A,B:o} true(A \wedge B) \rightarrow true(B) \\
\vee_{il} & : & \Pi_{A,B:o} true(A) \rightarrow true(A \vee B) \\
\vee_{ir} & : & \Pi_{A,B:o} true(B) \rightarrow true(A \vee B) \\
\vee_e & : & \Pi_{A,B:o} true(A \vee B) \rightarrow \Pi_{C:o}((true(A) \rightarrow true(C)) \\
 & & \rightarrow (true(B) \rightarrow true(C)) \rightarrow true(C)) \\
abs & : & \Pi_{A:o} true(\bot) \rightarrow true(A)
\end{array}
$$

Figure 1: An LF signature for intuitionistic logic

2.1 Judgements as Types

Logics are represented in LF by exploiting the *judgements-as-types* paradigm whereby judgements are associated with the type of their proofs (Harper *et al.*, 1992). In order to represent theories in LF, types of the framework are associated to each syntactic category of the language being represented (object language); then constants are declared to each expression-forming construct of the object language in such a way that a bijective correspondence between expressions of the object language and canonical forms of the right type is established.

In Figure 1, the syntactic category represented as LF type o corresponds to the type of propositions. The signature also contains types for the connectives $\supset$, $\wedge$, $\vee$ and constant $\bot$.

LF has only one type constructor: Π. It is a dependent function type constructor abbreviated as $\rightarrow$ when the range is independent of the domain. Π is used to express universal quantification and implication.

In LF, types are used to represent judgements and their inhabitants correspond to their proofs. So, rather than using the type system to represent formulae under the propositions-as-types paradigm (Martin-Löf, 1980), a judgement constructing function is declared in order to build basic judgements (called *atomic*). In the signature in Figure 1, the constant *true* is such a function. An atomic judgement corresponding to that signature is for example:

$$true(a \supset (a \supset b)))$$

The type constructor of LF allows us to specify two more kinds of judgement: hypothetical (e.g. $P \rightarrow Q$ with P and Q judgements) or schematic (e.g. $\Pi_{x:o}P(x)$ with o a type and P a judgement). Type constructors associate to the right; for example, $\Pi_{a:o}\Pi_{b:o}P \rightarrow Q$ is equivalent to $(\Pi_{a:o}(\Pi_{b:o}(P \rightarrow Q)))$.

Judgements may represent theorems of the logic such as:

$$\Pi_{A:o}\Pi_{B:o} true(A \supset (A \supset B) \supset B) \qquad (1)$$

where A and B are abstractions in type o which is defined to be the type of propositions. $true$ is a judgement valued function from o into $Type$, the kind that contains all types. Schematic judgement 1 states that it is true that

$$A \supset (A \supset B) \supset B$$

for all propositions A and B. It can be read as an axiom schema. Judgements may also be used to represent inference rules like:

$$\Pi_{A:o}\Pi_{B:o} true(A) \rightarrow true(A \supset B) \rightarrow true(B)$$

which states that if A and $A \supset B$ are true, then B will also be true for all propositions A and B. The usual format of inference rules is:

$$
\begin{array}{lcl}
\Pi_{A_1:o} \cdots \Pi_{A_n:o} J_1(A_1, \ldots, A_n) & \rightarrow & \cdots \\
 & \rightarrow & J_m(A_1, \ldots, A_n) \rightarrow K(A_1, \ldots, A_n)
\end{array}
$$

where J_i and K are judgement functions. We call Π_{A_i} the *quantification part* of the rule; we call $J_i(A_1, \ldots, A_n)$ the judgements in the *body* of the rule and $K(A_1, \ldots, A_n)$ the *head* of the rule.

In Figure 1, there are inference rules for implication elimination and introduction. Notice that inference

rules depend on other type definitions in the signature. Given a signature for a specific logic, LF can be used as a meta theory to set up theorems of the logic and verify them by testing the corresponding judgement, as a type, for *inhabitance*.

3 Proof Plans

We use *Proof Plans* as a paradigm to develop proof-automation techniques. Proof Plans consists of using a planner to build a proof plan out of methods. Methods allow us to separate the proof building procedure from the reasoning required to select a proof technique. Writing methods amounts to specifying when a proof technique should be used as opposed to verifying if the technique is applicable, which is what a proof editor does when it tries to apply a tactic to the current proof state.

Building proof plans from methods has the advantage of not having to do all the operations needed to apply the tactics. The architecture of the methods enables a declarative specification of the heuristics for the tactics while doing heuristic reasoning. This way, it is easy to edit and experiment with the heuristic information without altering the tactics themselves.

The methods we present in this work are designed for proof planner called MiniClam which is related to Clam (van Harmelen *et al.*, 1993). The proof plans built by MiniClam are applicable in a version of the Edinburgh Logical Framework programmed in the Mollusc proof editor (Richards *et al.*, 1994).

We present a setup whereby using proof plans we can reason about proofs in a framework theory by interpreting inference rules as rewrite rules and using rippling and other techniques to automate the process of isolating connecting expressions[1](see Section 4 below).

This approach is based on the principle of *difference reduction* described in Section 5. Difference reduction provides a pattern to develop different search strategies. By alternating difference unification and control techniques (such as rippling) to reduce differences, we can build general methods to plan proofs in frameworks theories. A Proof Plan in our setting will be composed mainly of methods that apply various difference reducing techniques based on rewriting. These methods will use rewrite rules extracted from the inference rules of the particular logic represented in the framework.

We take a new approach to generic proof search. We develop a system that obtains guidance from the object level logic by extracting rewrite rules from each logic presentation. The rewriting process is guided by a general strategy based on a special type of unification called *polarised coloured difference unification* (see Section 8).

[1]The notion of connecting expressions is taken in the sense of the Connection Method by Bibel (Bibel, 1982)

The system obtained is parameterised by the logic presentations of the framework theory and is therefore able to do a more specialised search guidance in each logic than a system based on uniform search.

In our system, the techniques used and the rewrite rules extracted define a different (but related) search space than the one induced by the original problem. Search in the new space is more tractable and proofs are easier to find.

4 Connecting Expressions

When doing search in framework theories, we often find that we want to produce *connections* between hypotheses and conclusions (i.e. identical expressions on both sides of the sequent) to obtain axioms. The word connection here is taken by analogy to the Connection Method (Bibel, 1982). In this method, complementary formulae (connection) are identified in a matrix-based description of the conjecture. Theoremhood there is defined in terms of paths of connecting formulae through the matrix. In order to obtain such *complementary* expressions in our work, we first look for them as subexpressions of the current hypotheses and goals at some step of a proof. Once we have located the two expressions that might make a connection, we start applying inference rules that isolate the desired formulae in the appropriate side of the sequent.

Connecting formulae are identified by a polarity value assigned to each expression. Polarity values can be $+$,$-$ and $\pm$ and are assigned by a special algorithm (see Section 7). Only unifying expressions with different polarity values constitute a potential connection.

5 Difference Reduction

Logics are usually represented in proof editors as a set of axioms and inference rules. Inference rules are the tools to convert one proof state into another and axioms just tell us when to stop —when building a proof backwards (from theorem to axioms)— or where to start —when building it forwards (from axioms to theorem).

In our system, proofs are built backwards from conjectures to axioms (i.e. sequents —in the sense of the Sequent Calculus— where the formula on the right-hand side (i.e. the goal) occurs in the left-hand side (i.e. the context)). To obtain axioms from the conjecture, one must decompose it by applying inference rules backwards. Each inference rule application transforms one proof state into another and expands the proof tree. Guiding search in a proof consists of deciding what inference rule is the most appropriate to transform every state into another one closer to an axiom.

The proof plans built by our method are based on rewriting operations as well as single inference rule applications. The operations are tactics that apply the

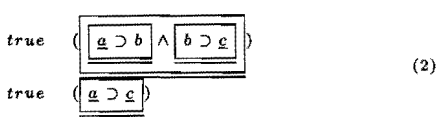
appropriate inference rules to produce the desired effect in the proof.

In order to select each rewriting operation the system uses the principle of *Difference Reduction* (Basin and Walsh, 1996b) which consists of two steps:

1. Compare two expressions using a special unifier called *difference unifier* (Basin and Walsh, 1993) to obtain annotations over the expressions. These annotations indicate the parts of the expressions that would have to be removed (the difference) in order to unify them.

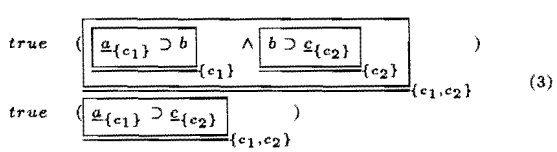
2. Use rewrite rules to successively rewrite the differing expressions in such a way that the difference between the two is reduced at each step.

Annotations consist of boxes which mark the differring parts and underlining that marks unifying subexpressions. The boxed expressions that are not underlined are called *wave fronts*; the underlined expressions that are not boxed form the *skeleton* set. These annotations may be combined to stress the unifying as well as the differring substructures between two terms. For example, the following are two difference-unified terms:



(2)

In these two expressions, the skeleton set is {true(a),true(c)}. We can also separate the two terms in the skeleton set and consider them as part of two separate skeleton sets: {true(a)} {true(c)}. In order to represent this on the expressions themselves, we imagine the skeletons to have colours and we annotate colour sets next to the wave fronts:



(3)

Here, c_1 and c_2 are meant to be colours. The unification algorithm that produces coloured (and polarity) annotations is called *Polarised Coloured Difference Unification.* It is described in Section 8.

6 Rippling

To monitor the movement of the difference, indicated in the original expression by the annotations, we annotate the rewrites as well. The annotations in the rewrites describe how the difference moves through the rule and help us to select the rules that will move the difference in the right direction. This kind of rewrite rule is called *wave rules* and the rewriting process involving wave rules is called *rippling.* The process continues until no more difference reduction can be done. Annotations also permit the definition of a measure of annotated terms. Wave rules are all by construction measure-decreasing rewrite rules and hence rippling is guaranteed to terminate (Basin and Walsh, 1996a).

Since we are interested in transforming a sequent into an axiom, we difference-unify the goal and the hypotheses of sequents. The rewriting operations selected by the planner are those that reduce the difference between the goal and the hypothesis closest to it in terms of similarity.

Rippling selects wave rules by matching their left-hand sides with the target expression in the usual way for rewrite rules; however, matching with annotations guarantees that the appropriate structures of the target expression will be shifted. Successive rewriting with wave rules may lead to the complete elimination of the difference between induction hypotheses and conclusion and hence to an axiom. This form of rewriting is more constrained than the unannotated kind and increases the chances of attaining a desired state.

7 Polarity

An important concept in our work is that of polarity. We use polarity annotations on framework theory expressions to make sure that rewriting operations in the plan correspond to sound inference rule applications in the proof editor. We also annotate object-logic expressions with polarity annotations derived from the corresponding signature heuristic to restrict difference unification to focus only on expressions that are likely to produce axioms. The use of polarity in this way is a novel technique. The algorithm that assigns polarity to subexpressions of formulas is described in detail in (Negrete, 1996).

In Section 4 we mentioned that connecting expressions are identified by their polarities and the assignment of polarity also plays a role in the selection of wave rules. The assignment of polarity values to subexpressions depends on the particular logic a formula corresponds to.

Inference rules may be used to refine the goal or may be applied forward to hypotheses. For this reason, from each inference rule two rewrite rules may be extracted: one that corresponds to the application of the inference rule left-to-right (forward-chain) and one right-to-left (refine).

We require wave rules to rewrite subexpressions of goals or hypothesis. For this reason, we need to be able to foresee which side of the sequent a subexpression potentially belongs to in order to know what wave rules are applicable to it. The simplest cases are the constructors

of the framework theory. In LF, a goal is assigned a positive polarity; we mark this with a positive sign as a superscript as follows:

$$\vdash (j_1^- \rightarrow j_2^+)^+$$

Inference rules are applied to goals right-to-left, so rewrite rules obtained from them will have positive polarity to match a goal. Since the constructor $\rightarrow$ can be introduced in such a goal leaving j_1 to the left of $\vdash$ and therefore inference rules are applicable to it left-to-right, j_1 is assigned negative polarity. When used as a hypothesis, the polarities are reversed:

$$(j_1^+ \rightarrow j_2^-)^- \vdash G$$

The cases for constructor Π are similar:

$$\vdash \Pi_{x:o^-} P(x)^+$$

and

$$\Pi_{x:o^+} P(x)^- \vdash G$$

The polarity values for o are added for completeness but are not used in practice, so we will not write them from now on. This way, an expression whose polarity is negative can be rewritten by a wave rule that encodes a forward-chaining operation. Conversely, positive polarity allows rewriting that encodes backward-chaining.

We also annotate expressions at the object level. This way, we can make a finer analysis to find connections at the object level. The potential connections can be simplified by rewriting until they become judgements of their own and are suitable for *fertilisation* (simplification of a sequent) as described in Section 9.

The assignment of polarity values depends on the encoding of the logic in the framework theory; different logics lead to different polarity assignments. We have developed an algorithm to assign polarity values to subexpressions of a formula with respect to a particular signature. Since the computation of these values may be computationally expensive, we do the assignment of polarity at the *comparison* stage explained in Section 9. When wave rules are extracted from the signature, they are also annotated to record the transition of polarities through rewriting. This means that rippling spreads both wave and polarity annotations through the planning process.

Our polarity algorithm assigns three polarity values to object level expressions: $+$,$-$ and $\pm$. The first two correspond to the ones described above for the framework level. The value $\pm$ means that the algorithm could not assign a definite value to the subexpression, either because the signature does not provide for it or because it has both positive and negative polarity. Two polarity values are *compatible* if they are not either two pluses or two minuses. See (Negrete, 1996) for a more detailed discussion of polarity and its properties.

Terms with polarity annotations are said to be *polarised*. Polarity values at the object level are used by the PCDU algorithm (described in the following section) to identify potential connections between two polarised terms. Two polarised terms with this property are said to be *compatible modulo polarity*.

8 Polarised Coloured Difference Unification

In this section we define formally the concepts of polarised annotated term and the algorithm for polarised coloured difference unification (PCDU) mentioned in Section 5. We first define what an annotated term is. Then, the concept of difference-unifiability is introduced. Finally, the full algorithm for PCDU is given. Again, to see the properties of the algorithm, the reader is referred to (Negrete, 1996).

8.1 Coloured-Annotated Terms

A polarised term is a term that has been assigned a polarity value; this is represented by a term of the form: t^-, t^+, $t^\pm$ where t is a regular term. We call a term algebra with polarity values assigned to all terms and subterms, a *free polarised term algebra*. We will use the function $pol(p)$ to refer to the polarity of the polarised term p and function $term(p)$ to refer to the term resulting from removing the polarity value from polarised term p.

The following is a definition of the set of polarised coloured annotated terms (*pcats*). We define the syntax of polarised terms with wave annotations:

Definition 1 *Let Σ be a signature; let $P = PTerm(\Sigma)$ be the free polarised term algebra over Σ, VP a set of variables and p a polarity sign; COL a set of colours. We inductively define a hierarchy of sets of coloured-annotated terms, AT_C, indexed by colour sets $C \subseteq COL$.*

- *if $X^p \in VP$ then $\underline{X^p}_C \in AT_C$.*
- *if $t^p \in P$ then $\underline{t^p}_C \in AT_C$.*
- *if $at_i \in AT_C$ then $\underline{f(at_1, \ldots, at_n)^p}_C \in AT_C$*
- *if $at_i \in AT_{C_i}$, $\bigcup_i^n C_i = C$, then $\underline{\boxed{f(at_1, \ldots, at_n)^p}}_C \in AT_C$*
- *Nothing else is in AT_C.*

Definition 2 *Given an annotated term as above, we can define what its skeleton is. The skeletons of a term are parameterised by colours. If c is a colour, we define:*

- $skel(\underline{t^p}_C, c) = \{\}$ *if* $c \notin C$
- $skel(\underline{X^p}_C, c) = \{X^p\}$ *if* $c \in C, X^p \in VP$.
- $skel(\underline{t^p}_C, c) = \{t^p\}$ *if* $c \in C, t^p \in P$.
- $skel(\underline{f(at_1, \ldots, at_n)^p}_C, c) = \{f(s_1, \ldots, s_n)^p | s_i \in skel(at_i, c)\}$ *if* $c \in C$.
- $skel(\underline{\boxed{f(at_1, \ldots, at_n)^p}}_C, c) = skel(at_1, c) \cup \ldots \cup skel(at_n, c)$ *if* $c \in C$.

Definition 3 *The function erase removes all annotation from an annotated term:*

- $erase(\underline{X^p}_C) = X^p$
- $erase(\underline{t^p}_C) = t^p$
- $erase(\underline{f(\vec{t_n})^p}_C) = f(\overrightarrow{erase(t_n)})^p$
- $erase(\underline{\boxed{f(\vec{t_n})^p}}_C) = f(\overrightarrow{erase(t_n)})^p$

The set $AT_C/t = \{s \in AT_C \mid erase(s) = t\}$ is the set of all annotated terms with the given erasure.

8.2 Polarised Coloured Difference Unifiability

As we saw in Section 5, polarised annotated terms are used to express differences in structure and proof-role of two terms. The wave annotations highlight the structural variance between them while the polarity values indicate the proof context in which they occur.

For our work, we need to identify terms which constitute potential connections. We therefore need to define our difference unification algorithm to unify terms which are structurally similar as in standard difference unification, but also whose skeletons have compatible polarities. We express this more formally in the definitions below.

The symbols $\circ$ and $\bullet$ represent compatible polarities: $\{+,-\}, \{+,\pm\}, \{-,\pm\}$ and $\{\pm,\pm\}$.

Definition 4 *The relation* $t_1 \doteq t_2$ *over polarised terms is true if* t_1 *and* t_2 *are equal modulo polarity compatibility. That is:*

1. $t^\circ \doteq t^\bullet$
2. $f(\vec{a_n})^\circ \doteq f(\vec{b_n})^\bullet \quad \forall i. a_i \doteq b_i$.

This definition is now extended to sets of polarised terms as follows:

Definition 5 *Two sets of polarised terms* P *and* Q *are compatible modulo polarity, expressed as* $P \rightleftharpoons Q$, *if:*

$P \rightleftharpoons Q$ *if* $(\forall p \in P. \exists q \in Q. p \doteq q) \wedge (\forall q \in Q. \exists p \in P. p \doteq q)$

The following definition states when two polarised terms are pcd-unifiable.

Definition 6 *Two polarised terms* t_1 *and* t_2 *are pcd-unifiable if there are two annotated terms* $at_1 \in AT_C/t_1$ *and* $at_2 \in AT_C/t_2$ *for some set of colours* C *and a substitution* τ *such that for all* $c \in C$

$$skel(at_1, c)\tau \rightleftharpoons skel(at_2, c)\tau$$

There may be more than one way in which terms may be pcd-unifiable. Just as in difference unification, there may be several pair of annotated terms which fulfill the requirements of Definition 6.

The algorithm presented in the next section computes, for any two terms t_1 and t_2, all variable substitutions that fulfill Definition 6. It has been adapted from the description of the algorithm for difference unification presented in (Basini and Walsh, 1993).

8.3 PCDU Algorithm

The following definition gives the rules for polarised coloured difference unification. The algorithm is defined as a non-deterministic set of transformation rules applicable to *triples* $\langle \sigma, S, \tau \rangle$. σ is a substitution of annotated terms for variables; we call it an *annotated substitution*. S is a sequence of tuples $\langle a, b, A, B \rangle$, called *pcdu-problems*. a,b are terms and A,B are variables where annotated terms will be incrementally instantiated (i.e. partial annotated terms with variables will be instantiated in them as new tuples are generated). We call sequence S the *problem sequence* of the triple. τ is a variable substitution of plain terms. We call τ the *term substitution* of the triple.

Given a colour set C, the algorithm starts with $\langle \{\}, \{\langle a, b, A, B \rangle\}, \{\} \rangle$ and ends with $\langle \sigma, \{\}, \tau \rangle$. $A\sigma$ and $B\sigma$ will be the annotated terms corresponding to a and b and τ will be the term substitution of the pcd-unification.

The algorithm always gives an answer. If two terms t_1 and t_2 don't difference-unify, the resulting annotated terms are: $\underline{t_1}_\emptyset$ and $\underline{t_2}_\emptyset$. This algorithm finds all common skeletons to the two terms and assigns them a colour; therefore, if no colour is assigned the terms are not difference unifiable.

The following definition gives a set of transformation rules. They take a triple —as defined above— and produce another one. The rules have the form:

$$T_1 \Rightarrow T_2 \text{ constraints: CONS}$$

, and denote the transformation of a triple matching T_1 into triple T_2 provided that $CONS$ hold. Constraints often rely on variables being instantiated in a state ahead of the present one, therefore, they have to be verified *post-hoc* when the information is available. They are only well-formedness constraints.

When the rules are applied exhaustively to a triple $\langle\{\},\{\langle a,b,A,B\rangle\},\{\}\rangle$ the final triple will contain the variable substitutions necessary to make a and b pcd-unifiable according to Definition 6. These rules are non-deterministic; the final triples given by all possible sequences of applications correspond to all possible pcd-unifications of a and b.

Definition 7 *If $at_1, at_2 \in AT_C/t$ for some t, then the function $superpose(at_1, at_2)$ is defined by pattern-matching:*

1. $superpose(\underline{t^p}_{C_1}, \underline{t^p}_{C_2}) = \underline{t^p}_{C_1 \cup C_2}$
2. $superpose(\underline{f(\overrightarrow{a_n})^p}_{C_1}, \underline{f(\overrightarrow{b_n})^p}_{C_2}) = \underline{f(\overrightarrow{superpose(a_n, b_n)})^p}_{C_1 \cup C_2}$
3. $superpose(\underline{\boxed{f(\overrightarrow{a_n})^p}}_{C_1}, \underline{\boxed{f(\overrightarrow{b_n})^p}}_{C_2}) = \underline{\boxed{superpose(f(\overrightarrow{a_n})^p, f(\overrightarrow{b_n})^p)}}_{C_1 \cup C_2}$
4. $superpose(t_1^p, t_2^p) = \underline{erase(t_1^p)}_{\emptyset}$

Definition 8 *In the following COL is a given set of colours, $c \in COL$, $C \subseteq COL$; p_1 and p_2 are atomic polarised terms, f is a function. A,B,$\overrightarrow{A_n}$ and $\overrightarrow{B_n}$ are annotated-term variables symbol. X and $\overrightarrow{X_n}$ are term variables. The symbols $\circ$ and $\bullet$ represent compatible polarities as above. The notation $S\{p_1,\ldots,p_n\}$ is used to represent the result of appending pcdu-problems $p_1,\ldots,p_n$ at the end of sequence S. We define the transformation rules for the algorithm as follows:*

- *DELETE*
  $\langle\sigma, S\{\langle a^\circ, a^\bullet, A_1, A_2\rangle\}, \tau\rangle \Rightarrow$
  $\langle\sigma\{\underline{a^\circ}_{\{c\}}/A_1, \underline{a^\bullet}_{\{c\}}/A_2\}, S, \tau\rangle$
  constraints: $constant(a)$, $select\_colour(c, COL)$.
- *DECOMPOSE*
  $\langle\sigma, S\{\langle f(\overrightarrow{a_n})^\circ, f(\overrightarrow{b_n})^\bullet, A, B\rangle\}, \tau\rangle \Rightarrow$
  $\langle\sigma\{\underline{f(\overrightarrow{A_n})^\circ}_C/A, \underline{f(\overrightarrow{B_n})^\bullet}_C/B\}, S\{\overrightarrow{\langle a_n, b_n, A_n, B_n\rangle}\}, \tau\rangle$
  constraints: $C = \overrightarrow{col(A_n)}$, $C = \overrightarrow{col(B_n)}$
- *ELIMINATE-L*
  $\langle\sigma, S\{\langle X^\circ, b^\bullet, A, B\rangle\}, \tau\rangle \Rightarrow$
  $\langle\sigma\{\underline{X^\circ}_{\{c\}}/A, \underline{b^\bullet}_{\{c\}}/B\}, S, \tau\{b/X\}\rangle$
  constraints: either $\{b/X\} \in \tau$ or $X \notin dom(\tau)$,$select\_colour(c, COL)$.
- *ELIMINATE-R*
  $\langle\sigma, S\{\langle a^\circ, X^\bullet, A, B\rangle\}, \tau\rangle \Rightarrow$
  $\langle\sigma\{\underline{a^\circ}_{\{c\}}/A, \underline{X^\bullet}_{\{c\}}/B\}, S, \tau\{a/X\}\rangle$
  constraints: either $\{a/X\} \in \tau$ or $X \notin dom(\tau)$,$select\_colour(c, COL)$
- *IMITATE-L*
  $\langle\sigma, S\{\langle X^\circ, f(\overrightarrow{b_n})^\bullet, A, B\rangle\}, \tau\rangle \Rightarrow$
  $\langle\sigma\{\underline{X^\circ}_C/A, \underline{f(\overrightarrow{B_n})^\bullet}_C/B\}, S\{\overrightarrow{\langle X_n, b_n, A_n, B_n\rangle}\}, \tau\{f(\overrightarrow{A_n})/X\}\rangle$
  constraints: $C = \overrightarrow{col(A_n)}$, $C = \overrightarrow{col(B_n)}$ and
  either $\{f(\overrightarrow{A_n})/X\} \in \tau$ or $X \notin dom(\tau)$.
- *IMITATE-R*
  $\langle\sigma, S\{\langle f(\overrightarrow{a_n})^\circ, X^\bullet, A, B\rangle\}, \tau\rangle \Rightarrow$
  $\langle\sigma\{\underline{f(\overrightarrow{A_n})^\circ}_C/A, \underline{X^\bullet}_C/B\}, S\{\overrightarrow{\langle a_n, X_n, A_n, B_n\rangle}\}, \tau\{f(\overrightarrow{B_n})/X\}\rangle$
  constraints: $C = \overrightarrow{col(A_n)}$, $C = \overrightarrow{col(B_n)}$ and
  either $\{f(\overrightarrow{B_n})/X\} \in \tau$ or $X \notin dom(\tau)$.
- *HIDE-L*
  $\langle\sigma, S\{\langle f(\overrightarrow{a_n})^\circ, b, A, B\rangle\}, \tau\rangle \Rightarrow$
  $\langle\sigma\{\underline{\boxed{f(\overrightarrow{A_n})^\circ}}_C/A, Sup/B\}, S\{\overrightarrow{\langle a_n, b, A_n, B_n\rangle}\}, \tau\rangle$
  constraints: $Sup = superpose(B_i)$, $C = col(Sup) \neq \emptyset$
- *HIDE-R*
  $\langle\sigma, S\{\langle a, f(\overrightarrow{b_n})^\circ, A, B\rangle\}, \tau\rangle \Rightarrow$
  $\langle\sigma\{Sup/A, \underline{\boxed{f(\overrightarrow{B_n})^\circ}}_C/B\}, S\{\overrightarrow{\langle a, b_n, A_n, B_n\rangle}\}, \tau\rangle$
  constraints: $Sup = superpose(A_i)$, $C = col(Sup) \neq \emptyset$
- *NOMATCH* $\langle\sigma, S\{\langle p_1, p_2, A, B\rangle\}, \tau\rangle \Rightarrow \langle\sigma\{\underline{p_1}_{\emptyset}/A, \underline{p_2}_{\emptyset}/B\}, S, \tau\rangle$

The function *col* returns the set of colours of an annotated term and is defined as:

$$col(\underline{t}_C) = C$$

In the *select_colour* relation, the first argument is a member of the set of colours in the second argument.

The function *dom* returns the domain of a substitution, that is, the set of variables to which terms are assigned.

9 Description of the Overall Methodology

Difference reduction provides a pattern to develop different search strategies. By alternating difference unification and control techniques to reduce differences, we can build general methods to plan proofs in frameworks theories. The preferred control technique is rippling because it is the most constrained. After rippling, unannotated rewriting is attempted. Finally, if the two previous options are not successful, inference rules are applied directly.

This hierarchy means that in the best cases, when only rippling steps are used, the search required to prove a theorem will be very small. In the places where rippling does not apply, the system may resort to more expensive steps to continue with the proof and try to resume rippling. This way, our system applies a well constrained methodology, like rippling, to produce proofs without much search but, when the methodology is not appropriate to a particular case, it *gracefully* degenerates into unconstrained search. From this point of view, our system is a hybrid approach to generic proof search guidance between specific systems with little search and scope, and uniform proof search methods which are very general but produce big search spaces.

As we said before, we use proof plans as a framework to implement our techniques. In the implementation in (Negrete, 1996), roughly each stage described below corresponds to a method for the Clam proof planning system (van Harmelen *et al.*, 1993):

Balancing When a proof is begun normally there are no hypotheses. They appear as the proof proceeds through applications of introduction rules. In order to

be able to obtain connections across the sequent symbol through Rippling, we need to justify the number of connections before starting the rewriting process. This is achieved by combining the application of introduction rules from the object logic and the framework theory to the conjecture. The object logic introduction rules cause the goal to fragment into smaller judgements linked by framework connectives and the framework rules introduce the new judgements into the hypothesis list. The process continues until a maximum number of potential connections is reached. This maximisation of potential connections is called balancing a sequent.

Comparison The second stage consists of difference unifying the goal and the hypotheses and ordering the set of annotated goal-hypothesis pairs. The order given to the set is induced by a measure of the difference between the members of the pair. This way, the members of the set of pairs will be selected in order.

Rippling The third stage consists of rippling both the goal and the selected hypothesis using wave rules extracted from the signature. Each time a wave rule is applied to the hypothesis the rewriting of it is reflected as a new hypothesis. The annotations are only kept on the last hypothesis so that a new rule may be applied to it.

Fertilisation The fourth stage consists of fertilising, that is, making a connection. The process consists of identifying connecting expressions in the sequent and reducing the sequent by making the connection. Fertilisation is usually possible after a successful rippling run. We have two ways of fertilising: backwards and forwards.

If one expression is the goal or is the head of the goal and the corresponding connecting expressions in the hypothesis is a hypothesis or the head of a hypothesis, then the connection can be made by backward-chaining the goal and the corresponding hypothesis.

For example, if the connection in the context is a hypothesis on its own, the sequent is trivial:

$$\cdots, j \vdash k_1 \rightarrow \cdots \rightarrow j$$

If the hypothesis containing the connection is a conditional judgement, then the hypothesis is used backwards as a derived inference rule to make the connection. We go from:

$$\cdots, l_1 \rightarrow \cdots \rightarrow j \vdash k_1 \rightarrow \cdots \rightarrow j$$

into:

$$\cdots, l_1 \rightarrow \cdots \rightarrow j, k_1, \cdots, k_n \vdash l_1$$

$$\vdots$$

$$\cdots, l_1 \rightarrow \cdots \rightarrow j, k_1, \cdots, k_n \vdash l_n$$

If one of the connecting expressions is part of the body of a hypothesis and the complementary expression is a hypothesis, then the connection is made by forward chaining. We go from:

$$\begin{array}{lcl} & \vdots & \\ hyp_1 & : & j \\ hyp_2 & : & l_1 \rightarrow \cdots j \rightarrow \cdots l_n \\ & \vdash & k \end{array}$$

to:

$$\begin{array}{lcl} & \vdots & \\ hyp_1 & : & j \\ hyp_2 & : & l_1 \rightarrow \cdots \rightarrow j \rightarrow \cdots l_n \\ hyp_3 & : & l_1 \rightarrow \cdots \rightarrow l_n \\ & \vdash & k \end{array}$$

hypothesis detail

After fertilisation, the branch of the plan is either complete or there are new sequents to solve. In the latter case the whole process is repeated.

Unblocking The system's strategy is to first reduce differences with wave rules because it is the most constrained way of reasoning about inference rule application. Not all rewrite rules parse into wave rules however. For this reason, if the application of wave rules fails, the unblocking stage tries to apply rewrite rules to unblock the rewriting process and go back to rippling. As before, not all inference rules translate into rewrite rules so, if also rewrite rule application fails, unblock tries the direct application of inference rules.

9.1 Analysing Logic Presentations

Logics can be represented in framework theories by defining the signature of a logic with the various constructors of the framework. In our system, we analyse logic presentations to extract rewrite rules and wave rules from them as was mentioned earlier.

The rewrite rules are extracted from the signature as follows:

1. For each inference rule, add rewrite rules corresponding to all the possible ways the inference rule can be applied forwards (called left-to-right rule or *lr-rule*) and backwards (called right-to-left rule or *rl-rule*), this process may produce *twin-rules* (see below).

2. Discard the rewrite rules whose left hand side is *unconstrained.*

3. Simplify the remaining rewrite rules where possible.

4. Assign positive polarity to both sides of rl-rules and negative polarity to both sides of lr-rules.

5. Polarise (i.e. add polarity annotations) to all subexpressions on both sides of the rewrite rules using the polarity algorithm.

Unconstrained rewrite rules are those whose left-hand-side is applicable to any or almost any expression. One of the usual constraints for rewrite rules in the literature is that their left hand sides are not variables. In the method we use to extract rewrites, there will never be variables in the left-hand sides of the rewrite rules. However, we still put a constraint on rewrite rules to avoid rewrites that are practically unconstrained. Those rules are the rewrite rules whose judgement in the left-hand side has a variable as argument. For example, after step 1 above we obtain rules like:

$$true(A) \Longrightarrow (true(A \supset B) \rightarrow true(B))$$

We avoid this kind of rule because they are too unconstrained.

The simplification of rewrite rules in Step 3 consists of transforming the rewrites obtained in the previous steps. This simplification step uses predetermined procedures to obtain optimised versions of rewrite rules that are more suitable for rippling. In the next section, when we introduce *non-standard rules*, we will see an example of such an optimisation for the $\vee_e$ rule.

From the inference rules of 1 we obtain the following set of rewrite rules[2]:

$$true(A^- \supset B^+)^+ \Longrightarrow (true(A)^- \rightarrow true(B)^+)^+ \qquad (\mathbf{rw}\text{-}\supset_i)$$

$$true(A^+ \supset B^-)^- \Longrightarrow (true(A)^+ \rightarrow true(B)^-)^- \qquad (\mathbf{rw}\text{-}\supset_e)$$

$$true(A^- \wedge B^-)^- \Longrightarrow true(A)^- \qquad (\mathbf{rw}\text{-}\wedge_{el})$$

$$true(A^- \wedge B^-)^- \Longrightarrow true(B)^- \qquad (\mathbf{rw}\text{-}\wedge_{er})$$

$$true(A^+ \vee B^+)^+ \Longrightarrow true(A)^+ \qquad (\mathbf{rw}\text{-}\vee_{il})$$

$$true(A^+ \vee B^+)^+ \Longrightarrow true(B)^+ \qquad (\mathbf{rw}\text{-}\vee_{ir})$$

$$true(\bot)^- \Longrightarrow true(A)^- \qquad (\mathbf{rw}\text{-}\bot)$$

They correspond to $\supset_i$, $\supset_e$, $\wedge_{el}$, $\wedge_{er}$, $\vee_{il}$, $\vee_{ir}$ and $\bot_e$ in that order. The versions of these rewrite rules in the opposite direction are unconstrained so they are discarded. Inference rules $\wedge_i$ and $\vee_e$ also produce rewrite rules but they are non-standard. We discuss these in the next section.

From the rewrite rules obtained we can now obtain wave rules as follows:

1. Use Polarised Coloured Difference Unification to annotate both sides of the rewrite.

2. Discard those wave rules which are not measure decreasing.

Following these steps we obtain from the rewrites above the following wave rules:

$$true(\boxed{\underline{A^-}_{C_1} \supset \underline{B^+}_{C_2}}_{C_3})^+ \Longrightarrow \boxed{\underline{true(A)^-}_{C_1} \rightarrow \underline{true(B)^+}_{C_2}}^{+}_{C_3} \qquad (\mathbf{wr}\text{-}\supset_i)$$

$$true(\boxed{\underline{A^+}_{C_1} \supset \underline{B^-}_{C_2}}_{C_3})^- \Longrightarrow \boxed{\underline{true(A)^+}_{C_1} \rightarrow \underline{true(B)^-}_{C_2}}^{-}_{C_3} \qquad (\mathbf{wr}\text{-}\supset_e)$$

$$true(\boxed{\underline{A^-}_{C} \wedge B^-}_{C})^- \Longrightarrow \underline{true(A)^-}_{C} \qquad (\mathbf{wr}\text{-}\wedge_{el})$$

$$true(\boxed{A^- \wedge \underline{B^-}_{C}}_{C})^- \Longrightarrow \underline{true(B)^-}_{C} \qquad (\mathbf{wr}\text{-}\wedge_{er})$$

$$true(\boxed{\underline{A^+}_{C} \vee B^+}_{C})^+ \Longrightarrow \underline{true(A)^+}_{C} \qquad (\mathbf{wr}\text{-}\vee_{il})$$

$$true(\boxed{A^+ \vee \underline{B^+}_{C}}_{C})^+ \Longrightarrow \underline{true(B)^+}_{C} \qquad (\mathbf{wr}\text{-}\vee_{ir})$$

The rule **rw-**$\bot$ cannot be converted into a wave rule because its two sides are not d-unifiable.

Weakening Coloured Wave Rules

The wave rules above are all coloured wave rules. They are used to ripple one or more colours (skeletons) at the same time. When two wave rules are equal as rewrite rules but the set of skeletons of one of them is a subset of the set of skeletons of the other one, we say that the wave rule with fewer skeletons is a *weakened* version (or a *weakening*) of the other one. There are cases where weaker versions of the wave rules originally computed from a signature are needed. These can be obtained by removing annotation corresponding to some skeletons from the wave rules as needed, with the condition that at least one skeleton remains. For example, a weakening of wave rule **wr-**$\supset_i$ is:

$$true(\boxed{\underline{A^-}_{C_1} \supset B^+}_{C_1})^+ \Longrightarrow \boxed{\underline{true(A)^-}_{C_1} \rightarrow true(B)^+}^{+}_{C_1}$$

[2] It is also possible to obtain rewrite rules from lemmata proved by the user. These could also produce useful wave rules. For example:

$$true(a \supset b) \rightarrow true(b \supset c) \rightarrow true(a \supset c)$$

produces twin wave rule (**wr-trans**):

$$true(\boxed{\underline{a^-}_{\{c_1\}} \supset \underline{c^+}_{\{c_2\}}})^+ \Longrightarrow \left\{ \begin{array}{l} true(\boxed{\underline{a^-}_{\{c_1\}} \supset b^+})^+ \\ true(\boxed{\underline{b^-}_{\{c_2\}} \supset c^+})^+ \end{array} \right.$$

9.2 Non-Standard Rules

We use some special kinds of rewrite rules which are different from the usual definition of rewrite rules. In this section we describe the characteristics which make them non-standard. We will only talk about rewrite rules but the same concepts extend to wave rules. Also, one rule can have more than one or all of the following non-standard characteristics.

Improper Rewrite Rules The first non standard rewrite rule that appears already in the list above, is rewrite rule **rw-⊥**. This rule has a variable in the right-hand side which does not appear in the left hand side. We call this kind of rule a *improper* rewrite rule (c.f. (Klop, 1992)). These rewrite rules introduce meta-variables in the proofs whose instantiation has to be deferred.

Twin Rewrite Rules Rule $\wedge_i$, when interpreted right-to-left in Step 1, is transformed into the rewrite rule:

$$(true(B) \rightarrow true(A \wedge B)) \Longrightarrow true(A)$$

This rule does not convey the meaning of the introduction of a conjunction, that is, "to prove a conjunction, it is necessary to prove each conjunct". We simplify this kind of rule in step 3 by creating a *twin-rule* (*twin rewrite rule*, *twin wave rule*). Twin-rules are non-standard rewrite rules that rewrite an expression in two different ways. The two ways are reflected in two copies of the original expression. We represent them using a key. For example, the twin rewrite rule corresponding to $\wedge_i$ is:

$$true(\boxed{\underline{A^+_{C_1}} \wedge \underline{B^+_{C_2}}}_{C_3})^+ \Longrightarrow \left\{ \begin{array}{l} \underline{true(A)^+}_{C_1} \\ \underline{true(B)^+}_{C_2} \end{array} \right. \quad (4)$$

This rule rewrites a goal of the form $true(A \wedge B)$ into two subgoals $true(A)$ and $true(B)$. This is exactly the effect produced by the original inference rule if used to refine a conjunction-goal.

The rule for $\vee_e$ is one of the rewrite rules that can be simplified in Step 3. This rule has a place-holder expression ($true(C)$) to match and preserve the goal while some hypothesis is eliminated. This type of rule is common in natural deduction style presentations of logics. The simplification of the rule consists of identifying this fact and converting the rule into one where the disjunction ($true(A \vee B)$) in a hypothesis (negative polarity) is rewritten into $true(A)$ and $true(B)$ as in the rule $\wedge_i$ mentioned above. Again we obtain a twin-rule:

$$true(\boxed{\underline{A^-_{C_1}} \vee \underline{B^-_{C_2}}}_{C_3})^- \Longrightarrow \left\{ \begin{array}{l} \underline{true(A)^-}_{C_1} \\ \underline{true(B)^-}_{C_2} \end{array} \right. \quad (5)$$

Context Rewrite Rules

Context rules (rewrite or wave) are rules where a new fresh variable is introduced in the rewritten term. The name of the variable depends on the context where the rewriting takes place. For example, the rule for existential elimination in natural deduction style predicate logic, encoded in LF as:

$$\Pi_{p:i\rightarrow o}\Pi_{q:o} true(\exists(p)) \rightarrow (\Pi_{t:i} true(p(t)) \rightarrow true(q)) \rightarrow true(q)$$

can be simplified as we did in the last section with rule $\vee_e$ to obtain the following rewrite rule:

$$true(\exists P^-)^- \Longrightarrow true(P^-(\hat{t}))^-$$

The expression $\hat{t}$ stands for a new variable fresh in the context at the time of the application of the rule. Variable $\hat{t}$ is generated when the rule is applied.

To see why this is needed we need to look at how the inference rule is applied. First, the inference rule is forward-chained with some hypothesis involving $\exists$, $\exists r$ say. This will generate a new hypothesis h_1 : $(\Pi_{t:i} true(r(t)) \rightarrow true(q)) \rightarrow true(q)$. Then this hypothesis is used to refine the goal, $true(z)$ say, and we obtain a new goal: $(\Pi_{t:i} true(r(t)) \rightarrow true(z))$. Finally, introducing Π and $\rightarrow$ we obtain the original goal $true(z)$ and a new hypothesis $true(r(\hat{t}))$ where $\hat{t}$ is a new variable of type i in the context.

Now, we will see two example theorems of how the whole system works.

10 Example Theorems and Proofs

This example is from propositional logic. We will use in the proof some of the wave rules introduced in the last section. The statement is:

Example 1 $true((a \supset b) \wedge (b \supset c) \supset (a \supset c))$

After providing polarity values, the system balances the sequent. Constants a and c make connections across the sequent symbol. Difference unification of goal and hypothesis gives us the wave annotation that mark the two skeletons (colours c_1 and c_2) that need to be rippled to make the connections:

$$hyp_1 \; : \; true(\boxed{\underline{a^+_{\{c_1\}}} \supset b^-}_{\{c_1\}} \wedge \boxed{b^+ \supset \underline{c^-_{\{c_2\}}}}_{\{c_2\}})^-_{\{c_1,c_2\}}$$
$$\vdash \; true(\underline{a^-_{\{c_1\}} \supset c^+_{\{c_2\}}})^+_{\{c_1,c_2\}} \quad (6)$$

The next step is to ripple the hypotheses. First, Rule **wr-$\wedge_{el}$** is applied:

$$
\begin{array}{lll}
hyp_1 & : & true(\boxed{(a^+ \supset b^-)^- \wedge \boxed{b^+ \supset \underline{c}^-_{\{c_2\}}}^-}^{\{c_2\}})^- \\
hyp_2 & : & true(\boxed{\underline{a}^+_{\{c_1\}} \supset b^-}^{\{c_1\}})^- \\
\vdash & & true(\boxed{\underline{a}^-_{\{c_1\}} \supset \underline{c}^+_{\{c_2\}}}^{\{c_1,c_2\}})^+
\end{array}
\tag{7}
$$

The annotations corresponding to the colour just rippled are removed from hyp_1. Now rule **wr-**$\wedge_{er}$ is applied to hyp_1 to ripple c_2:

$$
\begin{array}{lll}
hyp_1 & : & true((a^+ \supset b^-)^- \wedge (b^+ \supset c^-)^-)^- \\
hyp_2 & : & true(\boxed{\underline{a}^+_{\{c_1\}} \supset b^-}^{\{c_1\}})^- \\
hyp_3 & : & true(\boxed{b^+ \supset \underline{c}^-_{\{c_2\}}}^{\{c_2\}})^- \\
\vdash & & true(\boxed{\underline{a}^-_{\{c_1\}} \supset \underline{c}^+_{\{c_2\}}}^{\{c_1,c_2\}})^+
\end{array}
\tag{8}
$$

Rippling continues using a weakened version of Rule **wr-**$\supset_e$ on hyp_2:

$$
\begin{array}{lll}
hyp_1 & : & true((a^+ \supset b^-)^- \wedge (b^+ \supset c^-)^-)^- \\
hyp_2 & : & true(a^+ \supset b^-)^- \\
hyp_3 & : & true(\boxed{b^+ \supset \underline{c}^-_{\{c_2\}}}^{\{c_2\}})^- \\
hyp_4 & : & \boxed{\underline{true(a)^+_{\{c_1\}}} \rightarrow true(b)^-}^{\{c_1\}} \\
\vdash & & true(\boxed{\underline{a}^-_{\{c_1\}} \supset \underline{c}^+_{\{c_2\}}}^{\{c_1,c_2\}})^+
\end{array}
\tag{9}
$$

and then on hyp_3:

$$
\begin{array}{lll}
hyp_1 & : & true((a^+ \supset b^-)^- \wedge (b^+ \supset c^-)^-)^- \\
hyp_2 & : & true(a^+ \supset b^-)^- \\
hyp_3 & : & true(b^+ \supset c^-)^- \\
hyp_4 & : & \boxed{\underline{true(a)^+_{\{c_1\}}} \rightarrow true(b)^-}^{\{c_1\}} \\
hyp_5 & : & \boxed{true(b)^+ \rightarrow \underline{true(c)^-_{\{c_2\}}}}^{\{c_2\}} \\
\vdash & & true(\boxed{\underline{a}^-_{\{c_1\}} \supset \underline{c}^+_{\{c_2\}}}^{\{c_1,c_2\}})^+
\end{array}
\tag{10}
$$

At this point there is no more rippling possible in the hypotheses so rippling starts in the goal. The rule applied is **wr-**$\supset_i$:

$$
\begin{array}{lll}
hyp_4 & : & \boxed{\underline{true(a)^+_{\{c_1\}}} \rightarrow true(b)^-}^{\{c_1\}} \\
hyp_5 & : & \boxed{true(b)^+ \rightarrow \underline{true(c)^-_{\{c_2\}}}}^{\{c_2\}} \\
\vdash & & \boxed{\underline{true(a)^-_{\{c_1\}}} \rightarrow \underline{true(c)^+_{\{c_2\}}}}^{\{c_1,c_2\}}
\end{array}
\tag{11}
$$

Now both colours are fully rippled and fertilisation is possible with hyp_5 leaving the following goal:

$$
\begin{array}{lll}
hyp_4 & : & true(a)^+ \rightarrow true(b)^- \\
hyp_5 & : & true(b)^+ \rightarrow true(c)^- \\
hyp_6 & : & true(a)^- \\
\vdash & & true(b)^+
\end{array}
\tag{12}
$$

Fertilisation is again possible using hypothesis hyp_4:

$$
\begin{array}{lll}
hyp_4 & : & true(a)^+ \rightarrow true(b)^- \\
hyp_5 & : & true(b)^+ \rightarrow true(c) \\
hyp_6 & : & true(a)^- \\
\vdash & & true(a)^+
\end{array}
\tag{13}
$$

This sequent is an axiom and the proof is finished.

We present now one more example. This time from a different logic, in order to show how our techniques apply in other logics.

The modal logic system K from (Simpson, 1994) encoded in LF is:

$$
\begin{array}{rcl}
o, W & : & Type \\
true & : & W \rightarrow o \rightarrow Type \\
\bot & : & o \\
\wedge, \vee, \supset & : & o \rightarrow o \rightarrow o \\
\Box, \Diamond & : & o \rightarrow o \\
\\
\wedge_i & : & \Pi_{A,B:o}\Pi_{x:W} true(x,A) \rightarrow true(x,B) \rightarrow true(x, A \wedge B) \\
\wedge_{el} & : & \Pi_{A,B:o}\Pi_{x:W} true(x, A \wedge B) \rightarrow true(x,A) \\
\wedge_{er} & : & \Pi_{A,B:o}\Pi_{x:W} true(x, A \wedge B) \rightarrow true(x,B) \\
\vee_{il} & : & \Pi_{A,B:o}\Pi_{x:W} true(x, A) \rightarrow true(x, A \vee B) \\
\vee_{ir} & : & \Pi_{A,B:o}\Pi_{x:W} true(x, B) \rightarrow true(x, A \vee B) \\
\vee_e & : & \Pi_{A,B:o}\Pi_{x,y:W} true(x, A \vee B) \rightarrow \Pi_{C:o}((true(x,A) \rightarrow \\
 & & true(y,C)) \rightarrow (true(x,B) \rightarrow true(y,C)) \rightarrow true(y,C)) \\
\supset_i & : & \Pi_{A,B:o}\Pi_{x:W} (true(x,A) \rightarrow true(x,B)) \rightarrow true(x, A \supset B) \\
\supset_e & : & \Pi_{A,B:o}\Pi_{x:W} true(x, A \supset B) \rightarrow true(x,A) \rightarrow true(x,B) \\
\neg_i & : & \Pi_{A:o}\Pi_{x:W} (true(x,A) \rightarrow true(x,\bot)) \rightarrow true(x, \neg A) \\
\Box_i & : & \Pi_{A:o}\Pi_{x:W} (\Pi_{y:W}\ xRy \rightarrow true(y,A)) \rightarrow true(x, \Box A) \\
\Box_e & : & \Pi_{A:o}\Pi_{x:W}\ true(x, \Box A) \rightarrow xRy \rightarrow true(y,A) \\
\Diamond_i & : & \Pi_{A:o}\Pi_{x,y:W}\ true(y,A) \rightarrow xRy \rightarrow true(x, \Diamond A) \\
\Diamond_e & : & \Pi_{A,B:o}\Pi_{x,z:W}\ true(x, \Diamond A) \rightarrow (\Pi_{y:W}\ true(y,A) \rightarrow xRy \rightarrow \\
 & & true(z,B)) \rightarrow true(z,B)
\end{array}
$$

The judgement $true(x, A)$ in this signature means "proposition A is true in world x". The wave rules extracted from this signature are the following:

$$
true(x, \boxed{\Box \underline{A^+}_{C_1}}^{C_1})^+ \Longrightarrow \boxed{(\Pi_{y:W}\ (xRy)^- \rightarrow \underline{true(y,A)^+}_{C_1})^+}^{C_1} \quad (wr-\Box_i)
$$

$$
true(x, \boxed{\Box \underline{A^-}_{C_1}}^{C_1})^- \Longrightarrow \boxed{\Pi_{y:W}\ (xRy)^+ \rightarrow \underline{true(y,A^-)^-}_{C_1}}^{C_1} \quad (wr-\Box_e)
$$

$$
true(x, \boxed{\Diamond \underline{A^+}_{C_1}}^{C_1})^+ \Longrightarrow \left\{ \begin{array}{l} (xR_Y)^+ \\ \underline{true(Y, A^+)^+}_{C_1} \end{array} \right. \quad (wr-\Diamond_i)
$$

$$
true(x, \boxed{\Diamond \underline{A^-}_{C_1}}^{C_1})^- \Longrightarrow (xR\hat{y})^- \times \underline{true(\hat{y}, A^-)^-}_{C_1} \quad (wr-\Diamond_e)
$$

The symbols $\hat{y}$ and $\times$ have the same roles they had in the rewrite rules we described in the previous section. We can now move on to the proof of the following theorem:

Example 2 $true(x, \Box(a \supset b) \supset (\Box a \supset \Box b))$

As usual Balance is the first step:

$$hyp_1 \quad : \quad true(x, \boxed{\Box(\underline{a^+}_{\{c_1\}} \supset \underline{b^-}_{\{c_2\}})^-}_{\{c_1,c_2\}})^-$$
$$\vdash \quad true(x, \boxed{(\Box\underline{a^-}_{\{c_1\}})^- \supset (\Box\underline{b^+}_{\{c_2\}})^+}_{\{c_1,c_2\}})^+$$

rippling all skeletons first in the hypotheses (rules **wr-$\Box_e$** and **wr-$\supset_e$**) and then in the goal (rules **wr-$\supset_i$**, **wr-$\Box_e$** and **wr-$\Box_i$**) we get:

$$hyp_1 \quad : \quad true(x, \Box(a^+ \supset b^-)^-)^-$$
$$hyp_2 \quad : \quad \Pi_{y_1:W}(xRy_1)^+ \rightarrow true(y_1, a^+ \supset b^-)^-$$
$$hyp_3 \quad : \quad \boxed{\Pi_{y_1:W}(xRy_1)^+ \rightarrow \underline{true(y_1,a)^+}_{\{c_1\}} \rightarrow \underline{true(y_1,b)^-}_{\{c_2\}}}$$
$$\vdash \quad \boxed{(\Pi_{y_2:W}(xRy_2)^- \rightarrow \underline{true(y_2,a)^-}_{\{c_1\}})^- \rightarrow \Pi_{y_3:W}(xRy_3)^- \rightarrow \underline{true(y_3,b)^+}_{\{c_2\}}}_{\{c_1,c_2\}}$$

At this point **weak-fertilise** applies (using hyp_3):

$$hyp_3 \quad : \quad \Pi_{y_1:W}(xRy_1)^+ \rightarrow true(y_1,a)^+ \rightarrow true(y_1,b)^-$$
$$hyp_4 \quad : \quad \Pi_{y_2:W}(xRy_2)^+ \rightarrow true(y_2,a)^-$$
$$hyp_5 \quad : \quad (xRy_3)^-$$
$$\vdash \quad (xRy_3)^+$$

$$hyp_3 \quad : \quad \Pi_{y_1:W}(xRy_1)^+ \rightarrow true(y_1,a)^+ \rightarrow true(y_1,b)^-$$
$$hyp_4 \quad : \quad \Pi_{y_2:W}(xRy_2)^+ \rightarrow true(y_2,a)^-$$
$$hyp_5 \quad : \quad (xRy_3)^-$$
$$\vdash \quad true(y_3,a)^+$$

The first sequent is an axiom so method **axiom** closes the branch. The second sequent can be weak-fertilised with hyp_4 giving the following:

$$hyp_1 \quad : \quad true(x, \Box(a^+ \supset b^-)^-)^-$$
$$hyp_2 \quad : \quad \Pi_{y_1:W}(xRy_1)^+ \rightarrow true(y_1, a^+ \supset b^-)^-$$
$$hyp_3 \quad : \quad \Pi_{y_1:W}(xRy_1)^+ \rightarrow true(y_1,a)^+ \rightarrow true(y_1,b)^-$$
$$hyp_4 \quad : \quad \Pi_{y_2:W}(xRy_2)^+ \rightarrow true(y_2,a)^-$$
$$hyp_5 \quad : \quad (xRy_3)^-$$
$$\vdash \quad (xRy_3)^+$$

At this point, method **axiom** closes the branch.

Using all possible rule applications generates a combinatorial explosion. Our method reduces the number of rewrite rules applicable through the use of wave rules. Since potential connections are identified when hypotheses and conclusion are difference unified, rippling is likely to isolate them and close a branch. There is no absolute guarantee that this will be the case though, some times the combination of wave rule applications does not lead to connecting formulae being isolated. In these cases, backtracking is needed and big search may not be avoided with the current version of the methods.

Our methods need to be constrained further to account for these cases. At the moment, we only have experimental evidence that it is a good idea to use them and to continue developing them. They are also useful in that they show that it is possible to develop techniques that work across theories that can control search. More work needs to be done in the details of the methods to constrain further the cases when they don't work.

11 Conclusion

The development of framework theories has opened the possibility of representing formal systems in a uniform way. In these formalisms we can generalise current knowledge on proof automation to encompass a wide range of object theories. The work presented in this paper is an initial step to develop robust proof automation techniques applicable to as many theories as possible.

Effective proof search in framework theories is a hard problem. It requires selecting appropriate rules from the framework theory as well as the instantiations for these corresponding to the object-level rules. The applicability of framework rules is high because they implement abstract operations independent of the object logic —like variable instantiation (e.g. $\rightarrow l$) or term generalisation (e.g. $\rightarrow r$)— and hence are applicable to a large number of object rules. Object level rules change from theory to theory and present different shapes and uses. This makes it difficult to abstract a method to account for the way they are all used.

In this paper we have introduced a new approach to proof search in framework theories. The approach is based on difference reduction and proof plans. The contributions of our work stem from experience in designing and analysing a proof planning system for natural deduction style presentations of logics in the Edinburgh Logical Framework. We show that:

- Proof plans and difference reduction are promising paradigms to develop heuristics for proof search in framework theories.
- The extensions to the techniques of difference reduction we developed in this work improve the power of the existing techniques and raise important issues for the development of the theory of difference reduction.

Research on proof search guidance in framework logics has focused so far on uniform methods based on logic programming ideas (Helmink, 1991; Pym, 1990; Felty and Miller, 1991; Pfenning, 1991; Dowek, 1991). These methods guide search by exploiting powerful unification algorithms suitable to type and higher order theories and extend resolution ideas to framework logics.

Goal directed approaches to using signatures are difficult when these specify theories where introduction and elimination rules are involved. The problem is that elimination rules contain information related to the hypotheses and not to the goal. Any heuristic to apply them

needs to analyse the hypothesis list to select the right elimination rule, even in a backwards proof. This makes uniform search paradigms such as resolution difficult to control.

Our approach is different. First, we constrain search at the framework level by using tactics that implement basic meta-level operations (e.g. rewriting, refinement, elimination and introduction). The methods that specify these tactics contain declarative heuristics in their preconditions that control their applicability. In this setting, the number of choices for the system (i.e. the planner) is far less than the free selection of meta-level rules. Another advantage of this approach, inherited from proof planning, is that heuristics are localised, explicit and declarative so they are easily understood and modified.

In (Helmink, 1991), the rules of both the meta and object levels are transformed into Horn clauses to enable a goal directed search. As the proof proceeds, new entries in the hypothesis list are dynamically transformed into Horn clauses too. This method provides a uniform treatment of framework and object-logic rules under a single Prolog style goal directed search method but it also requires some tactical assistance to control search.

This system shares with our the idea of *compiling* the rules into a form more suitable to proof search than the original presentation. Helmink's system, however, is tied to backwards reasoning by the resolution mechanism whereas our system combines other mechanisms like forward chaining, balance, etc. Our system, on the other hand, lacks Helmink's system's ability to dynamically improve the rule data base. Our system could benefit from a similar approach by forming new wave rules as new objects enter the context.

In (Felty, 1989) object level theories are encoded in a subset of Higher Order Logic (hh^{ω}) and various tactical theorem provers based on λProlog (Miller and Nadathur, 1988)) are proposed for some logics. In (Felty, 1991) LF signatures are translated into hh^{ω} to take advantage of the goal directed search mechanism already developed for that theory.

In the system Elf (Pfenning, 1991), LF constructors are given a direct operational interpretation. An extension of Higher Order Unification is given in order to cope with dependent function types. Elf is a programming language where tactical theorem provers can also be programmed.

The resolution style mechanism in these systems is useful to develop tactics but it is not enough to guide proof search on its own. It is necessary to develop tactics to obtain a working theorem prover for a logic. It is possible to encode LF rules in λ-Prolog, as proposed by Felty (1991), and set a theorem to be proven by the Prolog-style mechanism of λ-Prolog using a logic variable to leave the proof object uninstantiated. This approach would work only for a few examples but, even for simple theorems such as [3]:

$$X : \Pi_{P,S:i\to o} true(\forall(\lambda_{x:i}.P(x) \wedge S(x)))$$

the number of definite clauses to use in backchaining is too big or the depth-first search mechanism of the language loops and does not produce a proof of the theorem.

There are examples of interactive theorem provers developed in λProlog in (Felty, 1989) but no generic automatic theorem prover has been reported in any resolution-style system where proof automation techniques have been developed for many logics.

Elf and Felty's systems can be used as object level provers for a planning system developed in λProlog with our methods. Our system puts emphasis on the planning system and use an object level prover designed in Prolog. The initial version of the planner we implemented is also built in Prolog but with the new λProlog it would be more natural to implement the new version of the planner and methods in this language to exploit its higher order syntax and unification capabilities. It is possible too, to use some of the ideas in (Felty, 1992) to implement the rewriting system of our methods.

Pym (1990) and Dowek (1991) also develop unification algorithm for type theories as basis for logic programming style search. Pym's work is on LF and Dowek's for the Calculus of Constructions.

The work just described takes the first step towards endowing framework logics with an operational mechanism to guide search. Helmink (1991) transforms the inference rules encoded in a framework logic to suit a Prolog style search mechanism; Felty (1989) proposes the Prolog style representation and mechanism as the framework itself; Pfenning (1991) adds a goal directed mechanism to the type theory to preserve its declarative properties.

References

Avron, F., and I. Mason, *Using typed lambda calculus to implement formal systems on a machine*, Report ECS-LFCS-87-31, Department of Computer Science, University of Edimburgh, July, 1987.

Basin, D., and T. Walsh, "Difference unification", in *Proceedings of the 13th IJCAI*, International Joint Conference on Artificial Intelligence, 1993. Also available as Technical Report MPI-I-92-247, Max-Planck-Institute für Informatik.

Basin, D., and T. Walsh, "Annotated rewriting in inductive theorem proving", *Journal of Automated Reasoning*, 16(1-2):147-180, 1996.

[3] We use standard notation instead of λ-Prolog's

Basin, D., and T. Walsh, "A calculus for and termination of rippling", *Journal of Automated Reasoning*, 1996. To appear 16(2-3), 1996.

Bibel, W., *Automated Theorem Proving*, Friedr. Vieweg & Sohn, Braunschweig/Wiesbaden, 1992.

Bundy, A., A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill, "Rippling: A heuristic for guiding inductive proofs", *Artificial Intelligence*, 62:185-253, 1993. Also available from Edinburgh as DAI Research Paper No. 567.

Bundy, A., F. van Harmelen, J. Hesketh, and A. Smaill, "Experiments with proof plans for induction", *Journal of Automated Reasoning*, 7:303-324, 1991. Earlier version available from Edinburgh as DAI Research Paper No. 413.

Constable, R. L., S. F. Allen, H. M. Bromley *et al.*, *Implementing Mathematics with the Nuprl Proof Development System*, Prentice Hall, 1986.

Coquard, Th., and G. Huet, "The calculus of constructions", *Information and Computation*, 76:95-120, 1988.

Dowek, G., *Démonstration Automatique dans le Calcul des Constructions*, Thèse de doctorat, Université Paris 7, décembre, 1991.

Felty, A., *Specifying and implementing theorem provers in Higher Order Programming Language*, PhD thesis, University of Pennsylvania, 1989.

Felty, A., *Encoding dependant types in an intuitionistic logic*, Technical Report 1521, INRIA, 1991.

Felty, A., "A logic programming approach to implementing higher-order term rewriting", in Eriksson, L-H *et al.*, editors, *Second International Workshop on Extensions to Logic programming*, Vol. 596 of *Lecture Notes in Artificial Intelligence*, pp. 135-161, Cambridge University Press, 1992.

Felty, A., and D. Miller, "Encoding dependent type λ-calculus in an intuitionistic logic", in Huet, G., and G. plotkin, editors, *Logical Frameworks*, pp. 215-251, Cambridge University Press, 1992.

Harper, R., F. Honsell, and G. Plotkin, "A framework for defining logics", *Journal of the ACM*, 40(1):143-84, 1992. Preliminary version in LICS'87.

Helmink, L., "Goal directed proof construction in type theory", in Huet, G., and G. Plotkin, editors, *Logical Frameworks*, CUP, 1991.

Klop, J. W., "Term rewriting systems", in Abramsky, S., D. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 2*, Vol 2, pp. 1-116. Claredon Press, Oxford, 1992.

Martin-Löf, P., *Intuitionistic Type Theory*, Bibliopolis, Naples, 1984. Notes by Giovanni Sambin of a series of lectures given in Padua, June, 1980.

Miller, D., and G. Nadathur, "An overview of λProlog", in Bowen, R., and K. Kowalski, editors, *Proceedings of the Fifth International Logic Programming Conference/ Fifth Symposium on Logic programming*, MIT Press, 1988.

Negrete, S., *Proof planning with logic presentations*, PhD thesis, Department of Artificial Intelligence, University of Edinburgh, May, 1996.

Pfenning, F., "Logic programming in the LF logical framework", in *Logical Frameworks*, pp. 149-182, Cambridge University Press, 1991.

Pym, D. J., *Proofs, search and computation in general logic*, PhD thesis, university of Edinburgh, 1990, Available as LFCS report ECS-LFCS-90-125.

Richards, B. L., I. Kraan, A. Smaill, and G. A. Wiggins, "Mollusc: A general proof development shell for sequent-based logics", in Bundy, A., editor, *12th Conference on Automated Deduction*, pp. 826-830, Springer-Verlag, 1994. *Lecture Notes in Artificial Intelligence.* Vol. 814. Also available from Edinburgh as DAI Research paper 723.

Simpson, A. K., *The Proof Theory and Semantics of Intuitionistic Modal Logic*, PhD thesis, Department of Computer Science, University of Edinburgh, 1994.

van Harmelen, F., A. Ireland, A. Stevens, S. Negrete, and A. Smaill, *The Clam proof planner, user manual and programmer manual (version 2.2)*, technical report, DAI, 1993.

Santiago Negrete *was awarded a BSc by Mexico's National University (UNAM) in 1990 and a PhD by the Department of Artificial Intelligence, University of Edinburgh in 1997. He worked in industry as research assistant in an Artificial Intelligence project in IBM-México (1988-90); he currently lectures at Instituto Tecnológico y de Estudios Superiores de Monterrey in Morelos and works as consultant for Infomedia S.A. de C.V. His research interests include formal methods, computational aspects of representation and reasoning with mathematical concepts, type theory and practical applications of them all.*