An Integration of Resolution and Natural Deduction Theorem Proving

Dale Miller and Amy Felty
Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104

Abstract: We present a high-level approach to the integration of such different theorem proving technologies as resolution and natural deduction. This system represents natural deduction proofs as λ-terms and resolution refutations as the types of such λ-terms. These type structures, called *expansion trees*, are essentially formulas in which substitution terms are attached to quantifiers. As such, this approach to proofs and their types extends the formulas-as-type notion found in proof theory. The LCF notion of tactics and tacticals can also be extended to incorporate proofs as typed λ-terms. Such extended tacticals can be used to program different interactive and automatic natural deduction theorem provers. Explicit representation of proofs as typed values within a programming language provides several capabilities not generally found in other theorem proving systems. For example, it is possible to write a tactic which can take the type specified by a resolution refutation and automatically construct a complete natural deduction proof. Such a capability can be of use in the development of user oriented explanation facilities.

1. Introduction

Theorem provers built on resolution and natural deduction have very different characteristics. For example, a search for a resolution refutation starts by taking a proposed theorem and putting its negation into skolem normal and conjunctive normal form. As a result of using such normal forms, the search space of refutations is very homogeneous, and automatic theorem provers using this paradigm are rather easy to build. On the other hand, since the search in such a theorem prover is carried out in a space which is rather remote from a user's original input, it is difficult to get the user to interact with the search process. On these accounts, natural deduction theorem proving is just the opposite. For example, no normal forms are generally used and only subformulas or instances of subformulas of the proposed theorem are used during the search for a proof. As a result, it is very easy to involve a user in the search for a proof since the state of the search at any moment is easily understood. On the other hand, natural deduction often leaves too many unimportant features in the search space which the preprocessing done by normal forms would have removed. Thus, resolution is often the core of automatic theorem provers while natural deduction is often the core of interactive theorem provers.

This work has been supported by NSF grants MCS8219196-CER, MCS-82-07294, and DARPA N000-14-85-K-0018.

Appearing in the Proceedings of the 1986 National Conference on Artificial Intelligence.

Clearly it is desirable to find some way to smoothly integrate these two very different paradigms. In this paper, we propose just such an integration. This integration is not a merging of the two different search spaces. It is, instead, an integration of the two kinds of proofs. We shall present a system which explicitly represents proofs in both systems and is capable of translating between them. In order to achieve this goal, we have designed a *programming language* which permits proof structures as values and types. This approach builds on and extends the LCF approach to natural deduction theorem provers by replacing the LCF notion of a *validation* with explicit term representation of proofs. The terms which represent proofs are given types which generalize the formulas-as-type notion found in proof theory [Howard, 1969]. Resolution refutations are seen as specifying the *type* of a natural deduction proofs. This high level view of proofs as typed terms can be easily combined with more standard aspects of LCF to yield the integration for which we are looking.

In Section 2 we describe a representation of natural deduction proofs as λ-terms, and in Section 3 we show how the LCF notion of tactics and tacticals can be used to specify an interactive theorem prover based on such a term representation of natural deduction proofs. In Section 4 we describe how resolution refutations can be converted to generalized type structures called *expansion trees.* In Section 5 we show how tactics can make use of the information stored in these generalized types. Also in Section 5, we present a program in the language of tactics which is capable of automatically converting a resolution refutation to a natural deduction proof.

2. Natural Deduction Proofs

Although much of what we describe here is applicable to most forms of natural deduction, the form we present in this paper is essentially the sequent system LK presented in [Gentzen, 1935] but without the cut rule. More modern presentations of similar systems can be found in [Gallier, 1986] and [Prawitz, 1965]. Proofs in the LK system are finite, ordered trees in which nodes are labeled with *sequents.* A sequent, written as $\Gamma \longrightarrow \Theta$, will represent the proposition "from all the formulas in the set Γ, some formula in the set Θ can be proved." Notice that the proposition connected to the sequent $A \longrightarrow A$ is trivially true. Sequents of this simple kind are called *axioms.* The non-terminal nodes of an LK proof are called inference rules and are listed below.

$$\frac{\Gamma \longrightarrow \Theta, A \qquad \Gamma \longrightarrow \Theta, C}{\Gamma \longrightarrow \Theta, A \wedge C}\ \texttt{and-r} \qquad \frac{A, C, \Gamma \longrightarrow \Theta}{A \wedge C, \Gamma \longrightarrow \Theta}\ \texttt{and-l}$$

$$\frac{A, \Gamma \longrightarrow \Theta \qquad C, \Gamma \longrightarrow \Theta}{A \vee C, \Gamma \longrightarrow \Theta}\ \texttt{or-l} \qquad \frac{\Gamma \longrightarrow \Theta, A, C}{\Gamma \longrightarrow \Theta, A \vee C}\ \texttt{or-r}$$

$$\frac{\Gamma \longrightarrow \Theta, A}{\sim A, \Gamma \longrightarrow \Theta}\ \texttt{not-l} \qquad \frac{A, \Gamma \longrightarrow \Theta}{\Gamma \longrightarrow \Theta, \sim A}\ \texttt{not-r}$$

$$\frac{\Gamma \longrightarrow \Theta, A \qquad C, \Delta \longrightarrow \Lambda}{A \supset C, \Gamma, \Delta \longrightarrow \Theta, \Lambda}\ \texttt{imp-l} \qquad \frac{A, \Gamma \longrightarrow \Theta, C}{\Gamma \longrightarrow \Theta, A \supset C}\ \texttt{imp-r}$$

$$\frac{[x/t]P,\Gamma \longrightarrow \Theta}{\forall x\, P,\Gamma \longrightarrow \Theta}\,\texttt{all-l} \qquad \frac{\Gamma \longrightarrow \Theta,[x/y]P}{\Gamma \longrightarrow \Theta,\forall x\, P}\,\texttt{all-r}$$

$$\frac{[x/y]P,\Gamma \longrightarrow \Theta}{\exists x\, P,\Gamma \longrightarrow \Theta}\,\texttt{some-l} \qquad \frac{\Gamma \longrightarrow \Theta,[x/t]P}{\Gamma \longrightarrow \Theta,\exists x\, P}\,\texttt{some-r}$$

$$\frac{\Gamma \longrightarrow \Theta}{A,\Gamma \longrightarrow \Theta}\,\texttt{thin-l} \qquad \frac{\Gamma \longrightarrow \Theta}{\Gamma \longrightarrow \Theta,A}\,\texttt{thin-r}$$

All but the last two rules are *introduction rules* and are responsible for introducing into sequents the various logical connectives. The proviso that the variable y is not free in any formula of the lower sequent must be added to the rules `all-r` and `some-l`. A derivation tree is an *LK proof of* A if the root of the tree is the sequent $\longrightarrow A$ and its leaves are axioms.

Example 1. Figure 1 is an LK proof of the formula

$$[p(a) \vee q(b)] \wedge \forall x\, [p(x) \supset q(x)] \supset \exists x\, q(x).$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{p(a) \longrightarrow p(a) \qquad \dfrac{q(a) \longrightarrow q(a)}{q(a) \longrightarrow \exists x\, q(x)}\,\texttt{some-r}}{p(a), p(a) \supset q(a) \longrightarrow \exists x\, q(x)}\,\texttt{imp-l}}{p(a), \forall x\, [p(x) \supset q(x)] \longrightarrow \exists x\, q(x)}\,\texttt{all-l} \qquad \dfrac{\dfrac{q(b) \longrightarrow q(b)}{q(b) \longrightarrow \exists x\, q(x)}\,\texttt{some-r}}{q(b), \forall x\, [p(x) \supset q(x)] \longrightarrow \exists x\, q(x)}\,\texttt{thin-l}}{p(a) \vee q(b), \forall x\, [p(x) \supset q(x)] \longrightarrow \exists x\, q(x)}\,\texttt{or-l}}{[p(a) \vee q(b)] \wedge \forall x\, [p(x) \supset q(x)] \longrightarrow \exists x\, q(x)}\,\texttt{and-l}}{\longrightarrow [p(a) \vee q(b)] \wedge \forall x\, [p(x) \supset q(x)] \supset \exists x\, q(x)}\,\texttt{imp-r}$$

Figure 1.

These proof trees can be represented more manageably as term structures. For example, let `axiom(A)` represent the proof tree which contains just the sequent $A \longrightarrow A$. The inference rules can be represented by function symbols of 1 or 2 arguments. For example, if T_1 and T_2 are proofs of $\Gamma, A \longrightarrow \Theta$ and $\Gamma, B \longrightarrow \Theta$, respectively, we would write `or_l`(T_1', T_2') to represent the proof

$$\frac{T_1 \qquad T_2}{\Gamma, A \vee B \longrightarrow \Theta}\,\texttt{or-l}$$

where T_1' and T_2' are the terms representing the proofs T_1 and T_2, respectively. Many inference rules require more information than just subproofs in order to put those subproofs together into larger proofs. For example, a term representing a proof which contains any of the quantifier introduction rules must contain the substitution term used to instantiate the quantifiers. Although such information is necessary, we avoid presenting it in this paper to simplify the presentation of examples.

Example 2. The (simplified) term which represents the proof in Example 1 is written as:

```
imp-r(and-l(or-l(all-l(imp-l(axiom(p(a)),
                                    some-r(axiom(q(a)))),
                      thin-l(some-r(axiom(q(b))))))).
```

To build interactive proof systems, it is important to represent not only completed proofs but also incomplete or partial proofs. We represent these by introducing into proof terms free variables which act as place holders for the actual subproofs. These free variables are also abstracted with λ-bindings. Thus a partial proof is represented as a function from subproofs to a completed proof.

Example 3. A partial proof of the formula in Example 1 is given in Figure 2 and by the term

```
lambda X lambda Y. imp-r(and-l(or-l(X,thin-l(Y)))).
```

$$
\cfrac{\cfrac{\cfrac{p(a), \forall x\,[p(x) \supset q(x)] \longrightarrow \exists x\, q(x) \qquad \cfrac{q(b) \longrightarrow \exists x\, q(x)}{q(b), \forall x\,[p(x) \supset q(x)] \longrightarrow \exists x\, q(x)}\ \texttt{thin-l}}{p(a) \vee q(b), \forall x\,[p(x) \supset q(x)] \longrightarrow \exists x\, q(x)}\ \texttt{or-l}}{[p(a) \vee q(b)] \wedge \forall x\,[p(x) \supset q(x)] \longrightarrow \exists x\, q(x)}\ \texttt{and-l}}{\longrightarrow [p(a) \vee q(b)] \wedge \forall x\,[p(x) \supset q(x)] \supset \exists x\, q(x)}\ \texttt{imp-r}
$$

Figure 2.

In order for the mechanism of λ-conversion to correctly represent the operation of supplying a partial proof with a subproof, we must *type* these λ-terms. For example, $\lambda x \lambda y\, T(x,y)$ represents a partial proof of some sequent in which two subproofs must be supplied. However, before this term can be applied to some actual proof, say S, one must check that the abstracted variable x is a place holder for proofs of the sequent for which S is a proof. Thus, we should make sequents and functions among sequents be the types of λ-terms. For example, if x and y are place holders for proofs of the sequents σ_1 and σ_2, respectively, and if $\lambda x \lambda y\, T(x,y)$ is a partial proof of the sequent σ, then we attach to this λ-term the type $\sigma_1 \rightarrow \sigma_2 \rightarrow \sigma$. The type of the λ-term representing the partial proof in Figure 2 is, therefore,

$$
\begin{aligned}
&(p(a), \forall x\,[p(x) \supset q(x)] \longrightarrow \exists x\, q(x)) \rightarrow (q(b) \longrightarrow \exists x\, q(x)) \rightarrow \\
&\qquad\qquad (\longrightarrow [p(a) \vee q(b)] \wedge \forall x\,[p(x) \supset q(x)] \supset \exists x\, q(x)).
\end{aligned}
$$

Remember that the typed λ-calculus has the following restriction on application: a term g can be applied to a term h if and only if the type of g is of the form $\alpha \rightarrow \beta$ and the type of h is of the form α. This restriction thus enforces the restriction of combining partial proofs with completed subproofs. We shall assume that the reader is familiar with the basic properties of λ-conversion.

3. Using Tactics to Build Proof Trees

The LCF system [Gorden, Milner, and Wadsworth, 1979] of tactics and tacticals can be easily extended to use the notion of proofs as typed λ-terms. In particular, tactics are functions which, when given a sequent (*i.e.* a type), either returns a partial proof for that sequent or fails. The main extension to LCF is that explicit representations of partial proofs are maintained through the use of λ-terms. In LCF, proofs and partial proofs are discarded as they are discovered.

Tactics are either primitive or compound. Primitive tactics attempt to prove a given sequent by using a particular inference rule. For example, the `or-l-tac` attempts to prove a sequent by using the `or-l` inference rule. A primitive tactic will fail if its own special inference rule is not applicable. If it succeeds, it returns a λ-term representing the partial proof which simply encodes that inference rule. If the tactic `or-l-tac` is applied to a sequent of the form $\Delta, A \vee C, \Gamma \longrightarrow \Theta$ it will succeed and return the λ-term

```
lambda X lambda Y. or-l(X,Y).
```

This term is typed as

$$(\Delta, A, \Gamma \longrightarrow \Theta) \rightarrow (\Delta, C, \Gamma \longrightarrow \Theta) \rightarrow (\Delta, A \vee C, \Gamma \longrightarrow \Theta).$$

This λ-term is a partial proof that stores a description of one step of the proof and represents the function which when given proofs of the types $\Delta, A, \Gamma \longrightarrow \Theta$ and $\Delta, C, \Gamma \longrightarrow \Theta$, would return a proof of the type $\Delta, A \vee C, \Gamma \longrightarrow \Theta$.

Compound tactics are built from primitive tactics by using tacticals. As in LCF, the then tactical is responsible for combining partial proofs. If we have a partial proof of type $\sigma_1 \rightarrow \sigma_2 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \sigma_0$, we need to compute proofs for each of $\sigma_1, \ldots, \sigma_n$ in order to have a complete proof of σ_0. Suppose we have decided to find a proof of the type σ_i, and some tactic or combination of tactics returns a partial proof of the type $\tau_1 \rightarrow \cdots \rightarrow \tau_m \rightarrow \sigma_i$. This is also a partial proof with m missing subproofs. The then tactical combines these two partial proofs into a single one of type

$$\sigma_1 \rightarrow \cdots \rightarrow \sigma_{i-1} \rightarrow \tau_1 \rightarrow \cdots \rightarrow \tau_m \rightarrow \sigma_{i+1} \rightarrow \cdots \rightarrow \sigma_n \rightarrow \sigma_0$$

which is a more refined partial proof of σ_0.

Example 4. Suppose that some combination of tactics returns the following partial proof:

$$\cfrac{\cfrac{p(a) \longrightarrow p(a) \qquad q(a) \longrightarrow \exists x\, q(x)}{p(a), p(a) \supset q(a) \longrightarrow \exists x\, q(x)}\ \texttt{imp-l}}{p(a), \forall x\, [p(x) \supset q(x)] \longrightarrow \exists x\, q(x)}\ \texttt{all-l}$$

where the term representing this partial proof is

```
lambda Z. all-l(imp-l(axiom(p(a)),Z)).
```

When this term is combined with the partial proof in Example 2, the combined proof can be written as

```
lambda Z lambda Y.
  imp-r(and-l(or-l(all-l(imp-l(axiom(p(a)),Z)),thin-l(Y))))
```

and is of type

$$(q(a) \longrightarrow \exists x\, p(x)) \rightarrow (q(b) \longrightarrow \exists x\, q(x)) \rightarrow$$
$$(\longrightarrow [p(a) \vee q(b)] \wedge \forall x\, [p(x) \supset q(x)] \supset \exists x\, q(x)).$$

Although the number of abstracted variables (*i.e.* the number of subproofs) may grow in size as we combine partial proofs, the amount of the proof that still must be completed generally decreases because as each rule is applied, the resulting sequent(s) generally contain fewer connectives. The number of subproofs decreases when one of them is recognized as an axiom.

In general, there are many terms (proofs) of a given type (sequent). Thus many choices can be made at each step in building a proof, and different choices can result in different proofs. These choices fall into two categories. The first choice at any given point in processing a partial proof is which abstracted variable (*i.e.* which subproof) should be analyzed. The second choice is which tactic to use in filling in this subproof. Tacticals allow the programmer to specify the order in which tactics are tried, *i.e.* control the order in which proof rules within the natural deduction system are attempted. For example, if we want to prove a set of theorems that we know all have the form $\longrightarrow (A_1 \wedge A_2 \wedge \ldots \wedge A_3) \supset B$ we may want to automate the part of the proof tree that breaks these connectives to get $A_1, A_2, \ldots, A_n \longrightarrow B$, then apply all non-branching propositional rules before continuing in interactive mode. A procedure to do this can be written as follows:

```
(then (then imp-r-tac (repeat and-l-tac))
      (repeat (orelse imp-r-tac neg-r-tac neg-l-tac
                      and-l-tac or-r-tac)))
```

where then, repeat, and orelse are names of high level tacticals similar to those found in LCF. By writing compound tactics, the programmer is directly involved in how choices are made during the search for a proof. The ability to express proof strategies as small programs allows great flexibility in customizing proof search and building proof heuristics.

Tactics look at the top-level connectives in sequents to determine which inference rules can be applied. We, however, have not discussed what happens when a top-level quantifier is encountered. The sequent itself does not have enough information to describe how that quantifer is to be introduced. Substitution information is required at this point. This information is not given by the sequent, and so the sequent by itself does not contain enough type information to adequately specify a proof. This type information is much harder to determine, and here we turn to an automatic theorem prover, such as resolution, for help.

4. Expansion Trees and Resolution Refutations

The substitution information which is lacking for this proof building process to continue could be supplied in a couple of ways. The search process could stop and ask the user for a substitution term. A more interesting possiblity, however, is to use a resolution theorem prover to supply this information since one of their strengths is the computation of substitutions via unification. The main question is how this information can be captured and used in the natural deduction setting. The problem of relating the substitution information in refutations to the building of natural deduction proofs is described in this section.

For concreteness, we first present a definition of resolution refutations. If B is a formula, let B^* denote its skolem normal form, *i.e.* essentially existential quantifiers are instantiated with Skolem terms and all essentially universal variables are deleted. We shall use $\mathrm{cnf}(B)$ to denote the set of sets of literals which comprises the conjunctive normal form of B. Let $\mathcal{H}_B$ denote the set of first-order terms which are composed only of functions and constants of B, plus an additional constant added to ensure that $\mathcal{H}_B$ is non-empty. A ***resolution refutation*** of B is a list of clauses (*i.e.* a set of literals) $C_1, \ldots, C_m$, such that C_m is the empty clause and for each $i = 1, \ldots, m$, one of the following is true:

(a) $C_i \in \mathrm{cnf}((\sim B)^*)$, or

(b) there are positive integers j, k less than i and sets of literals S_1 and S_2 such that $C_i = S_1 \cup S_2$, $C_j = S_1 \cup \{A\}$, and $C_k = S_2 \cup \{\sim A\}$, for some atomic formula A, or

(c) there is a substitution φ built using only terms in $\mathcal{H}_B$ and a positive integer $j < i$ such that $C_i = \varphi C_j$.

Example 5. The following is a refutation of the formula in Example 1.

(1)	$p(a), q(b)$	by (a)
(2)	$\sim p(x), q(x)$	by (a)
(3)	$\sim q(y)$	by (a)
(4)	$\sim p(a), q(a)$	by (c) from 2
(5)	$q(a), q(b)$	by (b) from 1 and 4
(6)	$\sim q(b)$	by (c) from 3
(7)	$q(a)$	by (b) from 5 and 6
(8)	$\sim q(a)$	by (c) from 3
(9)		by (b) from 7 and 8

Notice that refutations use substitutions in a very distributed fashion. Given a quantifier in the theorem it is not obvious what substitution terms were substituted for it. In contrast to refutations, we present another proof structure called ***expansion trees*** which store substitution information much more locally.

The tactics, as we described above, could not process a universal quantifier on the left or an existential quantifier on the right. This is because the sequents did not specify the instance of these expressions that should be used. We solve this problem by simply attaching to quantifiers in a formula what substitution instances are to be used during a proof. To this end, we define expansion trees and dual expansion trees in the following fashion.

(1) Let B be a formula. Then B is both an expansion tree and dual expansion tree for B.

(2) If Q_1 and Q_2 are expansion trees and Q_3 is a dual expansion tree for B_1, B_2, B_3, respectively, then the following are expansion trees for $B_1 \wedge B_2, B_1 \vee B_2, B_3 \supset B_2$, and $\sim B_3$, respectively: $Q_1 \wedge Q_2, Q_1 \vee Q_2, Q_3 \supset Q_2$, and $\sim Q_3$. This statement remains true if the role of expansion trees and dual expansion trees is switched.

(3) If y is a variable and Q is an expansion tree for $[x/y]B$ then $(\forall x\, B, (y, Q))$ is an expansion tree for $\forall x\, B$. If Q is a dual expansion tree for $[x/y]B$ then $(\exists x\, B, (y, Q))$ is a dual expansion tree for $\exists x\, B$.

(4) If $t_1, \ldots, t_n$ are first-order terms and for $i = 1, \ldots, n$, Q_i is an expansion tree for $[x/t_i]B$, then

$$(\exists x\, B, (t_1, Q_1), \ldots, (t_n, Q_n))$$

is an expansion tree for $\exists x\, B$. If, however, for $i = 1, \ldots, n$, Q_i is a dual expansion tree for $[x/t_i]B$, then

$$(\forall x\, B, (t_1, Q_1), \ldots, (t_n, Q_n))$$

is a dual expansion tree for $\forall x\, B$.

Example 6. An expansion tree for the formula in Example 1 is

$$[p(a) \vee q(b)] \wedge (\forall x\, (p(x) \supset q(x)), (a, p(a) \supset q(a))) \supset (\exists x\, q(x), (a, q(a)), (b, q(b))).$$

There are, of course, many other expansion trees for this formula.

We classify certain expansion trees as *expansion tree proofs* or simply *ET-proofs* if they satisfy two properties: one requires that a certain relation on the substitution terms in the tree be acyclic and the other requires that the "deep formula" represented by the tree be tautologous. For the definitions, the reader is referred to [Miller, 1984]. Roughly speaking, the deep formula of an expansion tree is simply a formula whose subformulas are taken from the terminal nodes of the tree. The deep formula of the expansion tree in Example 6 is

$$[p(a) \vee q(b)] \wedge [p(a) \supset q(a)] \supset [q(a) \vee q(b)].$$

Since this is tautologous, this expansion tree is in fact an ET-proof.

We now introduce a more general notion of type. A *generalized sequent*, written $\mathcal{P} \longrightarrow \mathcal{Q}$, contains a set of dual expansion trees $\mathcal{P}$ and a set of expansion trees $\mathcal{Q}$, such that the single expansion tree $[\wedge \mathcal{P}] \supset [\vee \mathcal{Q}]$ is an ET-proof.

The significance of expansion trees in the integration of natural deduction and resolution comes from the following two facts. First, a resolution refutation of a formula B can be converted to an ET-proof of B. Such an algorithm is presented in [Pfenning, 1984]. Here, skolem terms introduced by the refutation process must be removed. This, however, is very straightforward [Miller, 1983]. The refutation in Example 5 is converted by this procedure to the ET-proof in Example 6. Second, as shown in the next section, a generalized sequent can automatically be converted by a compound tactical to a natural deduction proof of the given type. There are, in general, many possible natural deduction proofs which could have this generalized sequent as their type, so such a conversion involves some searching. Here the search is concerned not with the existence of a proof but with the ***presentation*** of the proof. The search for different presentations can also be governed by compound tactics.

5. Expansions Trees-as-Types

Expansion trees can be viewed as types of LK proofs in the following sense. The sense in which formulas were types still applies since expansion trees generalize formulas. In addition, the substitution information in an expansion tree indicates how quantifiers within an LK proof get instantiated. For example, let Q be an ET-proof for A. An LK proof T of A is of type Q if the quantifier occurrences in T are introduced using the substitution terms attached to them in Q.

We now return to the description of building natural deduction proofs. Remember that tactics work by examining a type and suggesting a part of the proof which would build an element of that type. We have now introduced a more informative type structure. Hence, when the `all-l-tac` is called, there would be substitution terms attached to universal quantifiers on the left of the sequent. Such terms can, therefore, be used to do the required universal instantiation. The same is true for the other three quantifier rules. Hence, this new notion of type contains enough information to completely specify how to build a complete natural deduction proof. The following compound tactic performs exactly that operation.

```
(repeat
  (orelse (then thin-to-axiom axiomatize)
          and-l-tac imp-r-tac some-l-tac all-r-tac
          neg-l-tac neg-r-tac or-l-tac and-r-tac
          imp-l-tac or-r-tac some-r-tac all-l-tac))
```

There are many other compound tactics for building LK proofs of a generalized sequent. If this one is applied to the generalized sequent $\longrightarrow Q$ where Q is the expansion tree in Example 6 it would yield the natural deduction proof in Example 1, except that the `thin-l` rule would be swapped with the `some-r` rule.

It is possible to pair with expansion trees even more information to make for a richer type structure. For example, a type can be the triple $\mathcal{P} \longrightarrow \mathcal{Q}; \mathcal{M}$, where $\mathcal{P} \longrightarrow \mathcal{Q}$ is a generalized sequent, and $\mathcal{M}$ is a ***mating*** for the deep formula of $[\wedge \mathcal{P}] \supset [\vee \mathcal{Q}]$. Here, a mating is a graph of the literals of this formula which shows how various literals in it are connected. See [Andrews, 1980], [Andrews, 1981], [Bibel, 1981], and [Miller, 1984] for more on matings. By using matings, it is possible to make various tactics smarter. For example, it is possible to write a thinning tactic which can look "ahead" using the mating to determine that a certain formula in a sequent will never be needed in a certain subproof. The ability to throw away such formulas is very important for building coherent proofs. See [Miller, 1984] for more on using matings in this fashion.

6. Conclusions

Explicit representations of proofs provides this system with some capabilities not generally found in other theorem proving systems. Expansion trees can be used to store complete proofs in a very compact form. Proofs stored in such a form are also very flexible since they only represent a type of a natural deduction proof. Hence, when one wants to browse through or use such a proof in natural deduction form, there are many different presentations of it that can be made. Representing partial proofs as first-class values provides the ability to stop at any point in the proof process, and resume at a later time. The calculus of λ-conversion describes how partial proofs can be composed and the typing system is all that is needed for such compositions to be done soundly. This representation of proofs should also make it possible to implement many different kinds of algorithms *on* proofs which have been studied in proof theory. For example, one particularly exciting item to implement is the automatic conversion of proofs of a certain (constructive) kind to executable programs, such as is done in the PRL system [Bates and Constable, 1985].

There is very little about the LK proof system that is central to the development of this system. In fact, many different and less formal notions of natural deduction, such as natural language oriented explanations (see [Webber, Joshi, Mays, and McKeown, 1983]) could also be supported in many of the same ways we have discussed here.

Our current implementation of this system is built in a combination of Common Lisp and Prolog code. Besides being strongly related to LCF, much of the spirit of this implementation derives from the TPS system described in [Miller, Cohen, and Andrews, 1982].

7. References

[1] Peter B. Andrews, "Transforming Matings into Natural Deduction Proofs," *Fifth Conference on Automated Deduction, Les Arcs, France*, edited by W. Bibel and R. Kowalski, Lecture Notes in Computer Science, No. 87, Springer-Verlag, 1980, 281 – 292.

[2] Peter B. Andrews, "Theorem Proving Via General Matings," Journal of the Association for Computing Machinery **28** (1981), 193 – 214.

[3] Joseph L. Bates and Robert L. Constable, "Proofs as Programs," *ACM Transactions on Programming Languages and Systems,* Vol. 7, No. 1 (January 1985) 113 – 136.

[4] Wolfgang Bibel, "Matrices with Connections," Journal of the Association of Computing Machinery **28** (1981), 633 – 645.

[5] Jean H. Gallier, *Logic for Computer Science: Foundations of Automatic Theorem Proving*, Harper & Row, 1986.

[6] Gerhard Gentzen, *Investigations into Logical Deductions* in *The Collected Papers of Gerhard Gentzen* edited by M. E. Szabo, North-Holland Publishing Co., Amsterdam, 1969, 68 – 131.

[7] Michael J. Gorden, Arthur J. Milner, and Christopher P. Wadsworth, "Edinburgh LCF," Lecture Notes in Computer Science, No. 78, Springer-Verlag, 1979.

[8] W. A. Howard, "The formulae-as-type notion of construction," 1969. Published in J. P. Seldin and R. Hindley, ed. *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, 479 – 490, Academic Press, New York, 1980.

[9] Dale A. Miller, Eve Longini Cohen, and Peter B. Andrews, "A Look at TPS," *6th Conference on Automated Deduction, New York*, edited by Donald W. Loveland, Lecture Notes in Computer Science, No. 138, Springer-Verlag, 1982, 50 – 69.

[10] Dale A. Miller, "Proofs in Higher-order Logic," Ph. D. Dissertation, Carnegie-Mellon University, August 1983.

[11] Dale A. Miller, "Expansion Trees and Their Conversion to Natural Deduction Proofs," *7th Conference on Automated Deduction, Napa CA*, edited by R. E. Shostak, Lecture Notes in Computer Science, No. 170, Springer-Verlag, 1984, 375 – 393.

[12] Frank Pfenning, "Analytic and Non-analytic Proofs," *7th Conference on Automated Deduction, Napa CA*, edited by R. E. Shostak, Lecture Notes in Computer Science, No. 170, Springer-Verlag, 1984, 394 – 413.

[13] Dag Prawitz, *Natural Deduction*, Almqvist & Wiksell, Uppsala, 1965.

[14] Bonnie Webber, Aravind Joshi, Eric Mays, and Kathleen McKeown, "Extended Natural Language Data Base Interactions," *Computers and Mathematics with Applications* 9 (1983), 233 – 244.