DTP

Documentation

Donald F. Geddis

March 10, 1995

Contents

Chapter 1

DTP Overview

This manual describes the implemented inference engine called DTP[1] [Ged]. The system does refutation proofs of queries from databases in first-order predicate calculus, using a Model Elimination-style algorithm and domain-independent control of reasoning.

Technical discussion about the ideas behind some of the algorithms (particularily the caching strategies) can be found in my thesis [Ged95].

1.1 Introduction

Is DTP for you? The intended audience is those who need a reliable black box inference engine. DTP knows more about inference than most other theorem provers. An ideal application, for example, would be as the back end to a machine learning program or mobile robot. Such systems have a hard enough time just discovering true things about their worlds, much less figuring out how to arrange that knowledge in a computationally tractable way. The philosophy in DTP is that the user need only be concerned about writing down true axioms, and all search control knowledge will be embedded in the inference engine.

This style is not appropriate if the knowledge base constructor is also an expert in inference. For example, if the real intention in using the logic is to write a program, then perhaps PROLOG would be more appropriate. Similarly so if the user is willing to manipulate all their knowledge so that it may be expressed in a restricted form, *e.g.* as horn clauses. In that case, a much more efficient algorithm can be used which takes advantage of this metalevel constraint. Some examples of such systems are the aforementioned PROLOG, as well as OTTER [McC], the Boyer-Moore theorem prover [BM], RRL [Lab], FRAPPS [Fri], and the suite of algorithms in EPIKIT [GS].

1.1.1 Approach

How is one to deal with this task of solving arbitrary queries from an unknown knowledge base? It is clear that, since inference is NP-hard, not all queries will be answerable in a polynomial amount of time. The approach taken here is to include, as part of the overall system, algorithms that have the potential of eliminating some exponential part of the potential proof space, at a cost polynomial in the size of the space actually explored.

Some well known examples of such algorithms include tautology elimination and subsumption. In each case, the addition of the algorithms makes answering some queries slower (by a polynomial

[1] Don's Theorem Prover

amount of time), but it changes improves others from being unanswerable (by virtue of requiring an exponential amount of time) to actually returning an answer in polytime. Note also that the additional overhead is polynomial in the size of the explored space, *not* in the size of the database. So, for example, Pure Literal Elimination is done, but at the time of rule lookup, not prior to the query over the entire database.

1.1.2 Overview

DTP uses subgoaling inference with model elimination reductions, which makes the inference sound and complete. Additional pruning of the database results from pure literal elimination.

When solving conjunctions, backtracking is done via backjumping back to the nearest conjunct that bound some variable in the failed conjunct.

In addition, recursion control via caching and slaving (sometimes called "memoing") is implemented, so many axiom sets that appear to have infinite paths (*e.g.* a transitive rule) still result in a finite search space.

A residue-style assumption mechanism is integrated into the inference engine, allowing abductive reasoning to occur. A list of patterns may be specified; if a literal in the proof is an instance of one of the patterns, it may be assumed to be true. A second hook allows an arbitrary lisp function to be called in order to enforce consistency in the assumptions of a given proof effort.

Although function symbols are allowed, at the present time DTP performs neither term reasoning (with rewrite rules), nor does it have equality built in. The lisp arithmetic routines are interpreted, however, and ground terms with recognized function symbols will be simplified via procedural attachment. Term reasoning may be added by defining the function `term-inference` in the DTP package.[2]

Disjunctive answers are maintained through the use of answer literals.

The unification code was taken from Matt Ginsberg's MVL prover [Gin].

1.2 Syntax

Most of this manual assumes that the reader is familiar with the basic concepts of automated inference, such as resolution, unification, and database indexing. If not, many introductory AI texts (*e.g.* Ginsberg's [Gin93b]) contain at least an overview of the topic. More details of the standard approach can be found in a specialized text such as [GN87].

The logical syntax used by DTP is generally compatible with KIF, the Knowledge Interchange Format defined in [GF92]. Variables are lisp atoms beginning with the "?" character. Sentences are lisp s-expressions in prefix form, with the following operator constants predefined: `not`, `and`, `or`, `=>`, `<=`, and `<=>`. Many function constants are predefined to correspond to various arithmetic functions in lisp, as described in the section 1.4.7.

The operator semantics include the normal boolean interpretation of `and`, `or`, and `not` for conjunction, disjunction, and negation respectively.

$$\texttt{(<=}\ \alpha\ \beta_1\ \beta_2\ \ldots\ \beta_n\texttt{)}$$

means that α is true if all of the β_i's are true, and is thus equivalent to

$$\texttt{(or}\ \alpha\ \neg\beta_1\ \neg\beta_2\ \ldots\ \neg\beta_n\texttt{)}$$

[2]Vishal Sikka is exploring this approach. He may be contacted at `vishal@cs.stanford.edu`.

The sentence

(=> β_1 β_2 ... β_n α)

means the same as the previous one. The bidirectional sentence

(<=> α β)

means the same as

(and (<= α β) (=> α β))

All logical expressions in the database are converted internally into clausal form by the standard algorithm. Thus all existentially quantified variables are skolemized, and all remaining variables are universally quantified. A clause is merely a list of literals, each negated or not.

As an example, in logic one might write

$\forall x, y, z$ Outrun(x,z) $\subset$ Outrun(x,y) $\wedge$ Outrun(y,z)

or in PROLOG

```
outrun(X,Z) :- outrun(X,Y), outrun(Y,Z)
```

In DTP, this same sentence is typically written

```
(<= (outrun ?x ?z) (outrun ?x ?y) (outrun ?y ?z))
```

which is translated to the logically equivalent disjunctive list of literals

```
((outrun ?x ?z) (not (outrun ?x ?y)) (not (outrun ?y ?z)))
```

1.3 Inference

Subgoaling inference is a resolution-like rule which takes as input a literal and a clause and produces another clause. Let g be the subgoal literal and $\{l_1, \ldots, \overline{g_i}, \ldots, l_n\}$ be the literals in the clause, and assume that g unifies with $\overline{g_i}$ with the binding list σ. Then the resulting clause of the inference is $\{l_1, \ldots, l_{i-1}, l_{i+1}, \ldots, l_n\}|\sigma$, *i.e.* the same clause with the literal $\overline{g_i}$ removed and the binding list σ applied to all remaining literals.

For example, the subgoal `(outrun lion ?prey)`[3] resolves with the first literal in the clause

```
((outrun ?x ?z) (not (outrun ?x ?y)) (not (outrun ?y ?z)))
```

to produce the child clause[4]

```
((not (outrun lion ?y)) (not (outrun ?y ?prey)))
```

which is equivalent to the conjunctive subgoal

(outrun lion ?y) $\wedge$ (outrun ?y ?prey)

DTP applies this rule with the set-of-support restriction, so that subgoals are derived only from the initial query or some subsequent child clause. The result is conceptually similar to a combination of set-of-support and ordered resolution, where all contrapositives of each database sentence are also stored in the database. The inference (with the addition of reduction, described in section 1.3.1) is sound and complete for first-order logic.

[3]Which is equivalent to the clause `((not (outrun lion ?prey)))`.

[4]As usual, variables must be standardized apart within the database, so the variable `?y` appearing here must actually be replaced by some new variable not appearing elsewhere in the system.

1.3.1 Reduction

Reduction, a part of the model elimination procedure, is resolution between a subgoal and an ancestor of that subgoal, and it serves a function similar to factoring in pure resolution systems. It rarely comes up in practice, but is required for completeness.

Consider the database

```
(or (p a ?x) (p ?x a))
```

with the query `(p ?i ?j)`. Backward chaining on the goal by resolving with the first literal in the database rule leads us to `(not (p ?j a))` with `?i`→`a`. The reduction of this with the initial query (with binding `?j`→`a`) yields the answer `(p a a)`.

1.4 Enhancements

1.4.1 Iteration

DTP allows the inference space to be searched with depth-first iterative deepening (DFID) [Kor85] rather than simple depth-first search. This iteration may apply to either the depth of a subgoal, in terms of the length of the path from the root to the subgoal, or the depth of function nesting of terms in a subgoal.

This latter search is particularly important in DTP. Many infinite paths in the subgoal space can be dealt with using the recursion control techniques in section 1.5, but no such techniques are implemented for dealing with infinite function recursion. Thus standard approaches such as DFID are required in cases where such infinite spaces are likely.

An example using the `animal` theory follows. The sample run also takes advantage of postponement caching, as described in [Ged95]. The remainder of this section will assume familiarity with the caching example.

Figure 1.1 shows a trace of the query

```
(prove '(outrun lion ?prey) :all-answers t)
```

with the following variable settings changed from the defaults:

```
*theory*                  animal
*trace*                   (:cutoffs :iteration :answers)
*use-subgoal-cutoffs*     t
*initial-subgoal-depth*   1
*subgoal-depth-skip*      1
*subgoal-maximum-depth*   7
```

Some things to note about the trace include

1. Once an answer is found (at depth 2), and another is searched for, the search does not begin from scratch; rather, the current space is continued.

2. Repeated answers are detected and ignored. Naturally, the search to depth 3 finds (at least) the same answers at the one to depth 2; such answers are not reported.

3. The iteration only continues as long as some part of the space was pruned. In this case, despite a requested maximum depth of 7, the final phase was the search to depth 6, because no pruning occurred during that phase.

The proof completes, returning the usual multiple values

```
{((OUTRUN LION ZEBRA) (OUTRUN LION DOG) (OUTRUN LION (FOOD DOG)))
(NIL NIL NIL)
(NIL NIL NIL)
(#<Answer ?PREY->Zebra> #<Answer ?PREY->Dog> #<Answer ?PREY->Food(Dog)>)
#<Proof of (OUTRUN LION ?PREY) with 3 answers [Complete]>
```

1.4.2 Backjumping

When solving a conjunctive subgoal, most theorem provers do depth-first search through the space of solutions. The first conjunct is solved, the resultant bindings plugged in to the remaining conjuncts, and then the process iterates. When some intermediate instantiated subgoal has no solutions at all, then some form of backtracking must take place, to return to some previous choice point and try a different choice. In the typical depth-first search algorithms, this is implemented with chronological backtracking, returning to the most recently made choice in the event of a failure.

This search can be unnecessarily expensive, however, if the root cause of a failure at some conjunct is a poor choice at a very early conjunct. The same failure at the downstream conjunct will be discovered over and over again.

Problems very similar to this have been addressed in the constraint satisfaction literature. Their problems are different in at least two important ways:

1. The domains of the variables are finite, and known explicitly in advance.

2. In addition to the query, there is a set of constraints with nice properties, *e.g.* variable consistency can be checked against them with a very low complexity algorithm.

Solving a conjunction in inference doesn't share these properties. Nonetheless, it is often the case that the insights behind various constraint satisfaction algorithms can yield analogous algorithms in theorem proving. DTP uses a form of backjumping,[5] where the search backtracks past all (easily-computed) irrelevant conjuncts until locating one that actually impacts the detected failure. In essence, it is possible to discover short proofs that large portions of the search space will not have solutions either, given one particular failed solution.

DTP actually doesn't maintain the bindings themselves in the explanations for failure, merely the blamed conjuncts. (The bindings could be useful to avoid generating a new answer with the same failing bindings, but that's a rare occurrence anyway so the existing code avoids that additional complexity. See the end of this section for an example of how an algorithm might be even more clever.)

Example

An example may help illustrate backjumping in DTP.[6] Consider the query `G(?w)` for the database in table 1.1.

The proof space is shown in figure 1.2.

[5] Other candidate algorithms include GSAT, Min-conflicts, Dependency-directed backtracking, and Dynamic backtracking [Gin93a].

[6] The example presented only requires database lookup to solve. Note that this is only for clarity of explanation, as DTP does backjumping for full inference problems.

```
Subgoal depth cutoff = 1
Looking for next answer of (OUTRUN LION ?PREY)
Eat(Lion,?prey) cutoff because depth > 1 (subgoal max)

Subgoal depth cutoff = 2
Looking for next answer of (OUTRUN LION ?PREY)
Found answer #<Answer ?PREY->Zebra>

Subgoal depth cutoff = 2
Looking for next answer of (OUTRUN LION ?PREY)
Found answer #<Answer ?PREY->Dog>

Subgoal depth cutoff = 2
Looking for next answer of (OUTRUN LION ?PREY)
Outrun(Dog,?prey) cutoff because depth > 2 (subgoal max)
Eat(Zebra,?prey) cutoff because depth > 2 (subgoal max)
Carnivore(Lion) cutoff because depth > 2 (subgoal max)
Eat(Dog,?prey) cutoff because depth > 2 (subgoal max)

Subgoal depth cutoff = 3
Looking for next answer of (OUTRUN LION ?PREY)
Answer #<Answer ?PREY->Zebra> ignored because not new
Answer #<Answer ?PREY->Dog> ignored because not new
Eat(Dog,?prey) cutoff because depth > 3 (subgoal max)
Carnivore(Zebra) cutoff because depth > 3 (subgoal max)

Subgoal depth cutoff = 4
Looking for next answer of (OUTRUN LION ?PREY)
Answer #<Answer ?PREY->Zebra> ignored because not new
Answer #<Answer ?PREY->Dog> ignored because not new
Carnivore(Dog) cutoff because depth > 4 (subgoal max)

Subgoal depth cutoff = 5
Looking for next answer of (OUTRUN LION ?PREY)
Answer #<Answer ?PREY->Zebra> ignored because not new
Answer #<Answer ?PREY->Dog> ignored because not new
Found answer #<Answer ?PREY->Food(Dog)>

Subgoal depth cutoff = 5
Looking for next answer of (OUTRUN LION ?PREY)
Carnivore(Food(Dog)) cutoff because depth > 5 (subgoal max)

Subgoal depth cutoff = 6
Looking for next answer of (OUTRUN LION ?PREY)
Answer #<Answer ?PREY->Zebra> ignored because not new
Answer #<Answer ?PREY->Dog> ignored because not new
Answer #<Answer ?PREY->Food(Dog)> ignored because not new
```

Figure 1.1: Trace of Iterative-Deepening Proof

```
G(?w)   ⇐  A(?w) and B(?j) and C(?k) and D(?x) and
           E(?j,?k) and F(?w,?x)
A(1)
A(6)
B(2)
C(3)
C(5)
D(4)
E(2,5)
F(6,4)
```

Table 1.1: Illustration of Backjumping

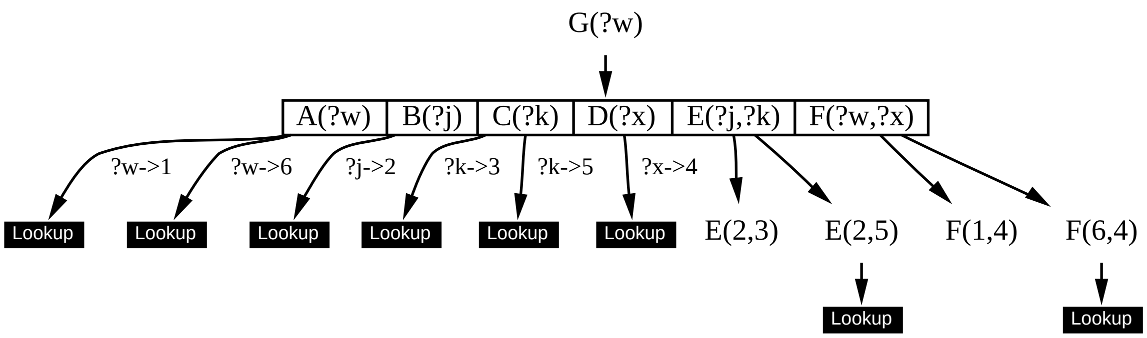

Figure 1.2: Illustration of Backjumping

We begin by solving the `A(?w)` conjunct, binding `?w` to 1. Then `B(?j)`, with `?j`→2, then `C(?k)`, with `?k`→3, and then `D(?x)` with `?x`→4.

At this point we attempt to solve `E(2,3)`, and fail, necessitating a backtrack. The explanations for the failure are `B` and `C`, the two previous conjuncts which bound variables appearing in `E`. We select the most recent one, `C`, and backjump to there, recording the remaining part of the explanation (conjunct `B`), in the set of explanations for `C`.

Continuing in the search, we call the `C(?k)` inferential generator again, yielding the new binding `?k`→5. Then `D` (having restarted, since we backed up over it) binds `?x` to 4 again, and this time `E(2,5)` succeeds with no additional bindings.

Now `F(1,4)` fails. Computing the explanations (*i.e.* the conjuncts which caused `?w` to be bound to 1 and `x` to 4), we get the set of conjuncts {`A,D`}. We back up to the most recent, namely `D`, and add conjunct `A` to `D`'s explanation set. Calling the generator on `D` yields no new answers, so we must back up again. No previous conjunct bound a variable in `D(?x)`, but from the failure at `F` we had already put conjunct `A` on `D`'s explanation set, so we backjump to `A` and search for a new answer.

The generator for `A(?w)` returns `w`→6 this time. The search from then proceeds much as before, including the rediscovery[7] of the failure to find a solution for `E(2,3)` and the backjump to `C` at that point. When we arrive at conjunct `F` the second time, we attempt to solve `F(6,4)`, and this succeeds. Thus the query is proved, with the bindings `?w`→6, `?j`→2, `?k`→5, and `?x`→4.

Improvement

Obviously, backtracking algorithms more complex than backjumping could be used. But there is a different, structural, limitation of DTP's current framework. All of the backtracking occurs within a conjunction, which is an object that is the child of some literal. It is impossible to jump across conjunctions.

When trying to prove the query

```
P(?x,?y) and Q(?x,?y)
```

from the database[8] in table 1.2, the conjunction

```
A(?x) and B(?y)
```

will be worked on as the child of `P(?x,?y)`, and

```
C(?y) and D(?x)
```

as the child of `Q(?x,?y)`. If the last child conjunct `D(?x)` fails, the best DTP can do is terminate that local conjunction, backtrack to the query conjunction, and ask for a new solution to `P(?x,?y)`.

The problem is that in effect we're working on the long conjunction

```
A(?x) and B(?y) and C(?y) and D(?x)
```

in which case a failure at `D(?x)` should cause us to backjump to `A(?x)`. Instead, DTP finds a new solution to `B(?y)` and then fails again on the conjunction below `Q(?x,?y)`, before finally generating a new answer to `A(?x)` and solving the query. (The proof space is shown in figure 1.3.) There of course could be an arbitrary amount of work involved in finding all the solutions to `B(?y)`, even though we can easily see that none of them will prevent a continuing failure for the subgoal `D(1)`.

[7] The attempt to avoid re-doing such work is the inspiration behind the dynamic backtracking algorithm [Gin93a].

[8] I am indebted to Michael Genesereth for this example.

```
P(?x,?y)  ⇐  A(?x) and B(?y)
Q(?x,?y)  ⇐  C(?y) and D(?x)
A(1)
A(2)
B(10)
B(11)
C(10)
C(11)
D(2)
```

Table 1.2: Backjumping across conjunctions

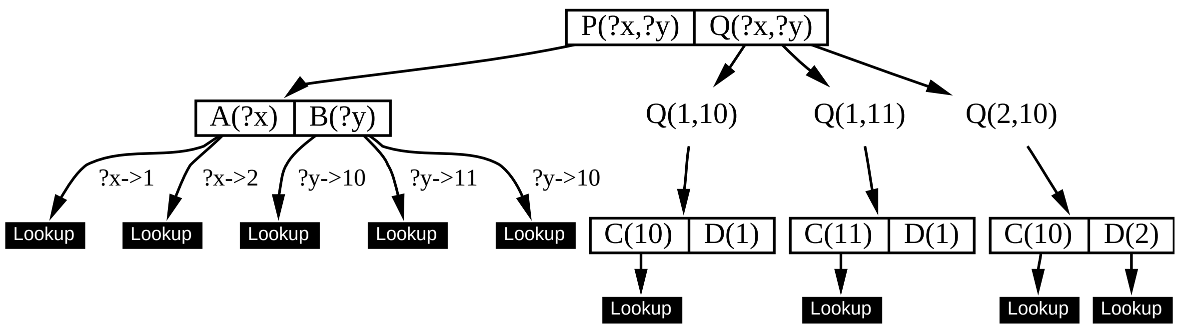

Figure 1.3: Backjumping across conjunctions

1.4.3 Disjunctive Answers

Consider the database composed of the single sentence

`Winner(Stanford) or Winner(Cal)`

A query of `Winner(?team)` will be interpreted as

Do you know that some team won the Big Game?

rather than as

Do you know of a particular team that won the game?

The returned answer will thus be the disjunction

`Winner(Stanford) or Winner(Cal)`

as shown in figure 1.4. (Note that this space shows a failed proof attempt, using a reduction with an ancestor goal. This path of reasoning does not result in an answer because the various binding lists on the path are not unifiable.)

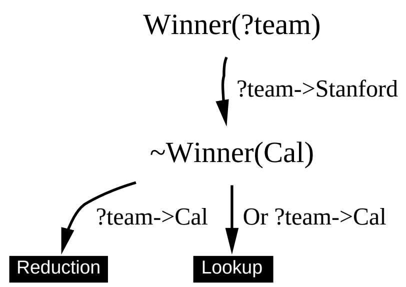

Figure 1.4: Disjunctive answer: Who won the Big Game?

The maintenance of the disjunctive binding lists is by a technique analogous to answer extraction of answer literals in resolution systems. The answer literals are attached to the negated goal (which must be added to the database during the proof to ensure completeness), so resolutions with the negated goal result in additional disjunctive bindings at the end.

1.4.4 Pure Literal Elimination

A pure literal is one which has no negated pair anywhere in the database.[9] Such literals can never be removed by refutation resolution, and thus clauses with pure literals can never participate in the proof of any query.

DTP does pure literal elimination. The implementation removes individual rules just before they are resolved against, rather than running a process over the whole database before the proof begins. (This allows the algorithmic complexity to be a function of the size of the proof, rather than the size of the database.)

Consider the database in table 1.3 with the goal `G`. Imagine that the subgoal `H` were very hard to prove, *e.g.* imagine that it reduced to solving Fermat's Last Theorem. Notice that the subgoal

G	$\Leftarrow$	H and I
G	$\Leftarrow$	P
H	$\Leftarrow$	Fermat
P		

Table 1.3: Pure Literal Elimination database

`I` is impossible to prove. DTP will recognize that the first rule is irrelevant, and thus will avoid the computationally expensive attempt to prove the subgoal `H`.

Here is a sample trace of such a proof:

```
Creating new subgoal G
Expanding subgoal G [0]
[Pure literal ~I detected in T-PURE-1...removing
       G <= H and I
 for duration of proof]
..Creating new subgoal P
Expanding subgoal P [1]
..Propagating #<Answer TRUE> to subgoal P
....Propagating #<Answer TRUE> to subgoal G
```

The proof space shown in figure 1.5 does not include any use of the first rule in the database.

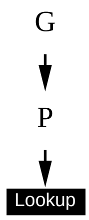

Figure 1.5: No pure literals

1.4.5 Residue/Abduction

A residue-style assumption mechanism is integrated into the inference engine. A list of patterns may be specified; if a literal in the proof is an instance of one of the patterns, it may be assumed to be true. Common examples of such a thing might be the various "not abnormal" predicates in many nonmonotonic descriptions of commonsense domains, or the existence of a component (like a digital gate in electronic design or an action is a planning domain) in a synthesis problem.

The assumption pattern list may be set by changing the value of the global variable `*assumables*`.

[9] Since resolving with the negated query is also a valid inference step, in order for a literal to be pure there must be no matching literal in the query either.

It is often the case that, while any individual assumption matching the pattern is a reasonable thing to assume true, that many combinations of assumptions are inconsistent. It is, of course, possible just to enumerate proofs containing assumptions and at the end discard those with inconsistent assumption sets, but such an approach is very inefficient. Thus a second hook allows an arbitrary lisp function to be called in order to enforce consistency among the assumptions in the middle of the proof effort itself.

The hook for the lisp function to check assumption consistency is the value of the global variable `*consistency-check*`.

1.4.6 Labels

The inference mechanism in DTP maintains labels on clauses, and combines them with local label computations. This isn't sufficient for more interesting kinds of reasoning (such as probability or fixed-point nonmonotonic systems) which have a global consistency check, but it nonetheless is occasionally useful. For example, MYCIN certainty factors and fuzzy logic both can be embedded within a local label system.

To use a new system of labels, a label structure must be defined within lisp. Required elements are: a minimum value, a maximum value, a conjunctive combination function, and a disjunctive combination function. The combination function must take as input two labels from the system, and return the label which is to be assigned to the `and` or `or` respectively of the input labels.

Some label structures are predefined in `labels.lisp`. For example, a version of qualitative likelihoods arises with the following Lisp code:

```
(defparameter *ql-values*
  '(false default-false unknown default-true true) )

(defun ql-and (&rest labels)
  (setq labels
    (mapcar
      #'(lambda (x) (position (label-value x) *ql-values*))
      labels ))
  (nth (apply #'min labels) *ql-values*) )

(defun ql-or (&rest labels)
  (setq labels
    (mapcar
      #'(lambda (x) (position (label-value x) *ql-values*))
      labels ))
  (nth (apply #'max labels) *ql-values*) )

(create-label-structure qualitative-likelihoods
 true false ql-and ql-or )
```

Note that no connection is made between the label for a sentence and the label for its negation. This allows the label computation to be a simple addition to the normal first-order proof space, but it also severely limits the type of label structures that can be used with DTP.

1.4.7 Procedural attachment

The common lisp arithmetic functions are procedurally attached to the theorem prover. Any ground terms encountered during the inference process, in which the function symbol is one of the following constants, is simplified by calling the lisp evaluator on the expression. Expressions that cause an error in the process lisp evaluation simplify to the constant 0.

The attached functions are:[10]

```
+ - * / 1+ 1- abs acos acosh ash asin asinh atan atanh boole ceiling cis
complex conjugate cos cosh decode-float denominator exp expt fceiling ffloar
float float-digits float-precision float-radix float-sign floor fround
ftruncate gcd imagpart integer-decode-float integer-length isqrt lcm log
logand logandc1 logandc2 logcount logeqv logior lognand lognor lognot logorc1
logorc2 logxor max min mod numerator phase rational rationalize realpart rem
round scale-float signum sin sinh sqrt tan tanh truncate
```

In addition, a ground term whose function constant is `eval` is simplified by applying the first argument to the rest of the arguments. Lisp errors result in a simplification to `NIL`. (The use of `eval` is intended to be a simple form of procedural attachment, so errors turn in to the generic lisp null value of `NIL`. The arithmetic functions above are typically used in the domain of numbers, which is why errors in those terms simplify to the usual arithmetic null value of zero.)

A hook exists in the code to extend the attachment mechanism. If the function `term-inference` is defined in the `DTP` package, then all ground terms not satisfying the first two conditions above[11] are replaced with the result of calling the lisp function `term-inference` on the ground term (which is a lisp list).

1.5 Caching

Caching is an attempt to reuse previous problem-solving effort, by replacing otherwise needed (possibly exponential) search with simple lookup. DTP offers a suite of possible caching schemes. Describing them all is beyond the scope of this section; please see my thesis [Ged95] for more details.

1.5.1 Conjunction Forking

Ideally, the caching mechanism would be independent of the heuristic question of what part of the space to search next. We would like to be able to explore relevant subgoals in any order that seems to be most useful.

Postponement caching in essence treats the search space as a true graph, rather than as the redundant tree typically explored by theorem provers. This means that when a solution is needed for a subgoal, say by a distant conjunction in the middle of searching its own space, a possible response is neither "here is a new answer," nor "there are no more answers" (which would let the conjunction backtrack), but rather "the final state is not known yet."

This means that the mechanism which searches the conjunction space must be capable of forking, allowing a subtree of the space to continue (under the condition that a distant subgoal finally finds

[10] This list was copied from EPIKIT's list of attached functions, after removing the special cases of `denotation`, `eval`, `execute`, `list`, and `name`.

[11] More precisely, this means terms with a function constant of either `eval` or one of the listed arithmetic functions.

a further answer), and also being able to continue the search under the assumption that an answer will never return.

Even that isn't sufficient, though. A deadlock cycle can arise, where all conjunctions of all subgoals are blocked, waiting for some other subgoal to return with some answer. Consider the database in table 1.4. The space in figure 1.6 shows the bug that can result if no special mechanism is implemented. The correct space (with a proof) is shown in figure 1.7.

`G`	$\Leftarrow$	`P(?x) and Q(?x)`
`P(1)`		
`P(2)`		
`Q(2)`		
`Q(1)`	$\Leftarrow$	`R and S`
`R`	$\Leftarrow$	`T and S`
`T`	$\Leftarrow$	`Q(1)`
`S`		

Table 1.4: Postponement blocks

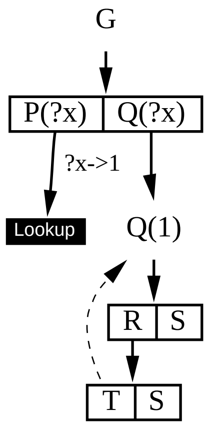

Figure 1.6: An apparently complete space

1.6 Future Work

1.6.1 Practical

The following ideas are missing from DTP not for any theoretical reason, but merely because of my own limited resources. Had I been the project leader for a team of programmers tasked to create an inference engine, I'd probably have assigned one to implement each of these concepts.

Lisp efficiency As this was a research tool, my implementation doesn't make the best use of efficiency hacks. In particular, non-destructive functions are used almost everywhere, even

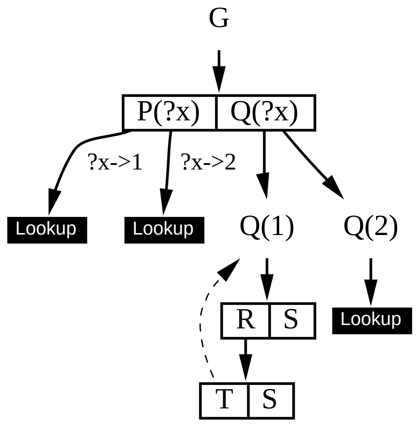

Figure 1.7: The actual complete space

when their destructive counterparts might be permissible. This makes the code much more reliable and easier to debug, but at a cost of loss of object code efficiency. In particular, binding lists are constantly copied (and literals are plugged in to), where more careful thought in the code probably would allow substantially less consing.

Equality Many applications of inference are most naturally stated with some form of equality, and many well-known algorithms address this issue. In particular, demodulation and paramodulation are probably useful things to have in the theorem prover.

Term reasoning In addition, full support for term rewrite rules probably makes sense. I suspect that all the same issues in inference (caching, recursion control) arise in an analogous way for term rewriting. It was because I didn't envision any *new* issues that this aspect of reasoning was left out of DTP.

Induction After term reasoning is implemented, it might be useful to allow inductive proofs on well-founded sets.

Fast database lookup State-of-the-art PROLOG systems are using Rete networks for fast matching and unification. *Alpha* nodes handle the matching of constants, and *beta* nodes take care of variables. The *alpha* network itself can be implemented with a discrimination net data structure.

Connection graph Clauses that resolve with each other can be cached, which also allows polytime forward inference in the graph.

Cache lookup Currently, a cache match on subgoals happens if the literals are identical up to variable renaming. There is an opportunity, however, to save some work if an existing subgoal is found that is an instance of the required subgoal. In that case, any current answers to the instance subgoal can be copied over to the newly created generalized subgoal. (Similarly, a new instance subgoal could check any existing generalized subgoals to see if they happen to contain appropriate answers already.)

Backtracking DTP uses backjumping to explore the search space of a conjunction, which is substantially better than the typical depth-first search of most inference algorithms. This could still be improved in two ways:

1. Use a more sophisticated constraint satisfaction algorithm, *e.g.* dynamic backtracking.
2. With any algorithm, one could record the failed binding itself in addition to the conjunct that had a failed binding. This way, when searching for a new answer to that conjunct, new answers which happen to have the same failed bindings could be rejected immediately.

1.6.2 Theoretical

Memory use

DTP's storage requirements grow linearly with time spent. The most blatant example is that the entire proof space that has been explored so far remains resident.

As Richard Korf writes [Kor92]:

> Since best-first search stores all generated nodes in the Open or Closed lists, its space complexity is the same as its time complexity, which is typically exponential. Given the ratio of memory to processing speed on current computers, in practice best-first search exhausts the available memory on most machines in a matter of minutes, halting the algorithm.

Thus search algorithms like iterative deepening have the advantage that even if an exponential amount of time is required to solve the problem, only linear space is necessary.

Rather than take an iterative approach,[12] DTP commits to vast storage requirements. The "correct" idea should be to use all the memory that is available at the time. When available memory is filled, selective forgetting can free up needed space, by throwing away the least useful part of the space. In order to maintain completeness, some form of summarization is needed. As an example in straight search, Korf's recursive best-first search algorithm [Kor92] maintains this selective forgetting with appropriate summaries.

I imagine something similar could work here, although it would require more theoretical effort to work out the details. At the moment DTP will just fill the available memory and then fail.

Counterexamples

When searching for a proof of some concept P, humans (after spending some effort and failing to discover a proof) will switch to trying to construct a counterexample for P, *i.e.* they'll spend some effort on a proof of $\neg P$. What is interesting about this from a theoretical point of view is that finding a proof for $\neg P$ typically aids you not at all in solving your top level goal; it merely means you must look elsewhere in the proof space for a solution to the original query. So the search for counterexamples is a purely metalevel computation.

What is the proper protocol for such a computation? How do you tradeoff some possible meta computation such as this one, with the additional progress you could make on the baselevel search? What is it, exactly, that you learn by the process of trying (and failing so far) to find a proof of some proposition? It what way does it lead you to believe more and more that the proposition is actually false?

[12] Not storing the space would force us to give up the other uses of the proof space, such as recursion control.

Decision Theory

Given a partial proof space with a fringe, which of the possible items on the fringe should be worked on next? It is difficult to apply decision theory directly, because the necessary information (expected computational cost and probability of success) is typically not available at the fringe of an inference proof space. What information *is* available? Is there a way to gather the needed decision theory information?

While recursion control takes care of many loops, it is still the case (especially with function symbols able to create an infinite number of objects) that a given tack might be able to be explored forever, without success. It seems again, as in the counterexample case, that you want some kind of exponential falloff in effort, that the longer you work on an approach without success, the more likely you'll try some other approach. Is there a formal basis for such a protocol? In what way is it different from doing breadth-first search, which seems to sacrifice focus?

1.7 Acknowledgements

The unification source code came from Ginsberg's MVL inference system [Gin].

Chapter 2

DTP User Manual

This appendix describes version 2.7 of DTP. Please send comments to Don Geddis at

    Geddis@CS.Stanford.EDU

or

Computer Science Department
Stanford University
Stanford, California 94305

The code was developed under Franz Allegro CL 4.2.beta.0 (on a Sun Sparc) and is written in Common Lisp with some CLtL2 extensions.[1] Earlier versions were tested under Lucid HP Common Lisp Rev. A.04.01 (on an HP-9000 Series 300/400) and MCL 2.0p2 (on an Apple Macintosh), although the latest version has not been.

DTP is available on the World Wide Web (WWW) from

    http://logic.stanford.edu/dtp/dtp.tar.gz

as a Unix-style TAR file, compressed with GNU's gzip utility.

The documentation is also available in the same DTP directory:

    http://logic.stanford.edu/dtp/manual.ps.gz

as a postscript file compressed with GNU's gzip utility.

2.1 Recommended Systems

There is an extensive explanation module built in to DTP. It is capable of producing text descriptions of proof spaces or justifications for answers, but the much clearer approach is to construct two-dimensional graphs. Such graphs are written in a format suitable for input to AT&T's DOT program. DOT is available by contacting Stephen North at

    north@research.att.com

[1]For example, liberal use is made of the `LOOP` macro.

The DOT program will convert the logical output of DTP into some metric form, for example postscript. While such files can of course be printed on a postscript-compatible printer, a better approach is to use a postscript previewer like the freely-available `ghostview` program for unix and X windows.

Finally, a large collection of theorem proving examples is available in the TPTP (Thousands of Problems for Theorem Provers) collection. It can be found on the World Wide Web at

`http://wwwjessen.informatik.tu-muenchen.de/~suttner/tptp.html`

2.2 Installation

Loading `dtp.lisp` loads all the other files. Customization is available by editing various forms within the file. In particular, logical pathnames for the system are defined there. Also, two functions (`compile-dtp` and `load-dtp`) are defined in `dtp.lisp`. It's generally a good idea to compile a lisp program before using it.

Installation options are implemented by modifying the `*features*` variable, which allows pieces of the code to have the compiler customize them. The available options are:

Customize

Feature `:customize-dfg` is present or absent. If present, it makes the default directories correct for Don Geddis.

Tracing

Feature `:dtp-trace` is present or absent. This controls whether to include code and data structures for watching the inference in the middle of problem solving (and for examining proof spaces afterwards). Most applications will want this option present, but some (*e.g.* an autonomous process) might wish the extra speed and smaller space, sacrificing user-friendliness.

Types

Feature `:dtp-types` is present of absent. If present, most functions will have the data types of their variables declared. Some Lisps react poorly to declaring types, and this feature may be removed without affecting the correctness of the code.

2.3 Examples

Most of the functionality is shown in the examples in the logical theories, which are exercised by running the function `(TEST-DTP)`. It is often useful to run some of those examples by hand, in particular after turning on full tracing output with a sequence like this:

```
(SETQ *THEORY* '<some theory>)
(SETQ *TRACE* *TRACE-KEYWORDS*)
(PROVE '<some goal>)
```

Other functions of interest include:

`(SETTINGS)` Describes the state of the theorem prover options.

`(POSSIBLE-SETTINGS)` Gives the valid possibilities for each of the variables in the `(SETTINGS)` list.

`(SHOW <object>)` Takes a proof object or an answer object and generates a postscript graph of the space using AT&T's program `dot`.

2.4 Demo

First load DTP by starting lisp and then evaluating the following forms:

```
(load "dtp")
(in-package "DTP")
(test-dtp)
```

(The last form should return "0", indicating no errors.)

To run examples by hand, try (for example):

```
(push :proofs *trace*)
(setq *theory* 'dtp2)
(show *theory*)
(prove '(a ?i ?j)) [Returns (A 0 9)]
(show-proof-graph *last-proof*)
(prove-next-answer *last-proof*) [Returns (A 1 2)]
(show-proof-graph *last-proof*)
(prove-next-answer *last-proof*) [Returns (A 6 5)]
(show-proof-graph *last-proof*)
(prove-next-answer *last-proof*) [Returns NIL]
(show-proof-graph *last-proof*)
```

An example that demonstrates recursion control is available by attempting to prove the query

```
(outrun lion ?prey)
```

in the theory `animal`, after setting `*caching*` to `:postponement`. See section 1.5 for an explanation of the proof.

2.5 Theory mechanism

The active database at any one time is defined by a root theory, and the transitive closure of the subtree of the theory DAG beginning at that root theory. All sentences in any theory within that subtree are database sentences for the current proof.

Chapter 3

DTP Reference Manual

3.1 Variables

The function `(settings)` will list all the parameters of the theorem prover, and show their current settings. A structured list of the set-able variables is shown in figure 3.1. In the list below, the first line of each entry is the variable, initial value, and grouping.

assumables `nil` *General*
: A list of patterns such that literals which are an instance of one of the patterns are assumed to be true, via the residue mechanism.

consistency-check `nil` *General*
: A lisp function, or NIL. If present, sets of literals assumed by the residue mechanism are forced to remain consistent according to this function.

display-color `nil` *Graphics*
: When constructing a graph, cache links between subgoals need to be distinguished from normal inference links. If true, the two classes of links are drawn with different colors. If false, inference links are drawn with solid arcs and cache links with dashed arcs.

display-landscape `t` *Graphics*
: Orientation of page for graphs. If true, graph is drawn and displayed on an 11" by 8.5" page. If false, the standard 8.5" by 11" is used.

display-logic-as-lists `nil` *Output*
: Output form preference. Whether literals should be printed with relation names before or after the parentheses, and whether constants should be capitalized. When `t`, literals look like `(father don jim)`; when `nil`, the appearance is `Father(Don,Jim)`.

display-one-page `t` *Graphics*
: Whether to try to compress a drawn graph to force it to appear on a single page. Also centers drawing on that page.

display-query `t` *Graphics*
: Whether to add a node at the top of a proof drawing with the original query. If true, this additional node will force all drawings to be a graph. If false, disjunctive queries may cause drawing to be a forest of disconnected graphs instead.

General

`*theory* *assumables* *consistency-check*`

Inference Engine

```
*use-negated-goal* *use-subgoal-inference* *use-contrapositives*
*use-reduction* *use-pure-literal-elimination* *use-backjumping*
*caching* *use-residue* *use-procedural-attachments* *use-reordering*
```

Iteration

```
*use-subgoal-cutoffs* *initial-subgoal-depth* *subgoal-depth-skip*
*subgoal-maximum-depth* *use-function-cutoffs* *initial-function-depth*
*function-depth-skip* *function-maximum-depth*
```

Display form

`*display-logic-as-lists* *show-renamed-variables* *graphic-display*`

Graphic Display

```
*display-one-page* *display-as-figure* *display-landscape*
*display-title* *display-query* *display-color*
*display-constrained-ranks*
```

Tracing

`*trace* *single-step*`

Figure 3.1: User Set-able Variables in DTP

display-title `t` *Graphics*
Whether to add descriptive text below a proof space drawing.

function-depth-skip `1` *Iteration*
The (integer) value to increment the function depth cutoff during every level of the iterative search.

function-maximum-depth `nil` *Iteration*
Subgoals with terms having nested functions greater than or equal to this value (if set to an integer) will be pruned in the search space. This is also the final function depth cutoff if iteration is occurring.

initial-function-depth `1` *Iteration*
When iterating the function depth cutoff, the is the initial cutoff.

initial-subgoal-depth `1` *Iteration*
When iterating the subgoal depth cutoff, the is the initial cutoff.

show-renamed-variables `nil` *Output*
When sentences are retrieved from the database, variables must all be renamed. It is often easier to read output with the original names, and the possible confusion of similar-printing but distinct variables rarely occurs. This variable controls whether the unique integer suffix is printed when the variables are output. (This variable has nothing to do with the underlying algorithms, which always use renamed variables.)

subgoal-depth-skip `1` *Iteration*
The (integer) value to increment the subgoal depth cutoff during every level of the iterative search.

subgoal-maximum-depth `nil` *Iteration*
Subgoals with terms having nested functions greater than or equal to this value (if set to an integer) will be pruned in the search space. This is also the final subgoal depth cutoff if iteration is occurring.

theory `global` *General*
Root theory for sentences to be used in proofs, if not specified in the `prove` function call. The set of sentences will be those in the transitive closure of the theory DAG beginning at this root.

trace `(:file-load :tests)` *Tracing*
A list of keywords, each of which controls the output of trace output printed as DTP runs. The possible options are in the list `*trace-keywords*`, and resetting DTP sets it to the value of `*trace-defaults*`. Its initial value is the value of `*trace-defaults*`.

use-backjumping `t` *Inference*
Backjumping (vs. chronological backtracking) upon failure in a conjunction.

use-caching `t` *Inference*
Caching and recursion control.

use-contrapositives `t` *Inference*

If the database is not Horn, then for ordered resolution (the basic inference mechanism) to be complete, all contrapositives of each database sentence must also be used. If α implies β, then the contrapositive of that rules is that the negation of β implies the negation of α.

use-function-cutoffs `t` *Iteration*

If true, the various function cutoff variables will be active and thus the search space will be pruned if the function nesting of terms in a subgoal is more than one of the bounds.

use-negated-goal `t` *Inference*

For completeness, the negated goal must be added to the database before a proof attempt begins.

use-procedural-attachments `t` *Inference*

Term simplification.

use-pure-literal-elimination `t` *Inference*

Eliminate rules with pure literals.

use-reduction `t` *Inference*

Goal-goal resolutions (a subgoal with an ancestor).

use-reordering `t` *Inference*

Reorder subgoal agenda, to put most likely subgoals first.

use-residue `t` *Inference*

Assumption mechanism.

use-subgoal-cutoffs `nil` *Iteration*

If true, the various subgoal cutoff variables will be active and thus the search space will be pruned if the length of the path from the root to a subgoal is more than one of the bounds.

use-subgoal-inference `t` *Inference*

The basic resolution mechanism (ordered resolution).

3.2 Functions

These functions are exported from the DTP package. In the list below, the first line of each entry is the function, arguments, and source file.

all-theories *hierarchy.lisp*

List all theory names with defined sentences. Related functions: `includes unincludes includees decludes included-active-theory-names show-theory-dag`.

brf `fact` *epikit-dtp.lisp*

Related functions: `knownp proval remval prologp prologx prologs save drop empty facts content`.

contents `theory` *epikit-dtp.lisp*

Simulate EPIKIT function to display the contents of a theory. Related functions: `knownp proval remval prologp prologx prologs save drop empty facts brf`.

decludes `theory-name` *hierarchy.lisp*
Related functions: `includes unincludes includees included-active-theory-names show-theory-dag all-theories`.

drop `fact theory` *epikit-dtp.lisp*
Simulate EPIKIT function to remove a fact from a theory. Related functions: `knownp proval remval prologp prologx prologs save empty facts contents brf`.

drop-sentence-from-theory `sentence &key :theory-name :test` *database.lisp*
Related functions: `empty-theory make-theory-from-sentences save-sentence-in-theory sentences-in`.

dtp-load `filename` *file.lisp*

empty `theory` *epikit-dtp.lisp*
Simulate EPIKIT function to empty a theory of facts. Related functions: `knownp proval remval prologp prologx prologs save drop facts contents brf`.

empty-theory `theory-name` *database.lisp*
Related functions: `make-theory-from-sentences save-sentence-in-theory drop-sentence-from-theory sentences-in`.

facts `atom theory` *epikit-dtp.lisp*
Simulate EPIKIT function to return list of clauses in a theory. Related functions: `knownp proval remval prologp prologx prologs save drop empty contents brf`.

included-active-theory-names `theory-name` *hierarchy.lisp*
Related functions: `includes unincludes includees decludes show-theory-dag all-theories`.

includees `theory-name` *hierarchy.lisp*
Related functions: `includes unincludes decludes included-active-theory-names show-theory-dag all-theories`.

includes `theory-name-1 theory-name-2` *hierarchy.lisp*
Related functions: `unincludes includees decludes included-active-theory-names show-theory-dag all-theories`.

knownp `fact theory` *epikit-dtp.lisp*
Simulate EPIKIT function to look up a fact in a theory (no inference). Related functions: `proval remval prologp prologx prologs save drop empty facts contents brf`.

load-logic-samples *test.lisp*
Load the sample logic files that come with the DTP system. Related functions: `test-dtp show-proofs`.

make-theory-from-sentences `theory-name sentence-label-pairs` *database.lisp*
Related functions: `empty-theory save-sentence-in-theory drop-sentence-from-theory sentences-in`.

plug `x bdg-list` *bindings.lisp*

prologp `fact theory` *epikit-dtp.lisp*

Simulate EPIKIT function to prove a fact from a theory. Related functions: `knownp proval remval prologx prologs save drop empty facts contents brf`.

prologs `expr fact theory` *epikit-dtp.lisp*

Simulate EPIKIT function to find all proofs of a fact from a theory. Related functions: `knownp proval remval prologp prologx save drop empty facts contents brf`.

prologx `expr fact theory` *epikit-dtp.lisp*

Simulate EPIKIT function to prove a fact from a theory (returning bindings). Related functions: `knownp proval remval prologp prologs save drop empty facts contents brf`.

proval `fact theory` *epikit-dtp.lisp*

Simulate EPIKIT function to find the value of a term. Related functions: `knownp remval prologp prologx prologs save drop empty facts contents brf`.

prove `query &key :all-answers :return-form :suppress-disjunctive-answers` *prover.lisp*

Related functions: `prove-next-answer proof-all-remaining-answers`.

prove-all-remaining-answers `&optional *proof*` *prover.lisp*

Related functions: `prove prove-next-answer`.

prove-next-answer `&optional *proof*` *prover.lisp*

Related functions: `prove proof-all-remaining-answers`.

remval `fact theory` *epikit-dtp.lisp*

Simulate EPIKIT function to remove the value of a term. Related functions: `knownp proval prologp prologx prologs save drop empty facts contents brf`.

reset-dtp *internals.lisp*

save `fact theory` *epikit-dtp.lisp*

Simulate EPIKIT function to save a fact in a theory. Related functions: `knownp proval remval prologp prologx prologs drop empty facts contents brf`.

save-sentence-in-theory `sentence &key :theory-name :label` *database.lisp*

Related functions: `empty-theory make-theory-from-sentences drop-sentence-from-theory sentences-in`.

sentences-in `theory-name &key :with-atom` *database.lisp*

Related functions: `empty-theory make-theory-from-sentences save-sentence-in-theory drop-sentence-from-theory`.

settings *output.lisp*

Displays the values of all user set-able global variables. Related functions: `show show-proof-graph`.

show `object` *view.lisp output.lisp*

`object` may be of type `proof`, `answer`, or `symbol`. In the first two cases, text output or a DOT graph is drawn, depending on the value of `*graphic-display*`. If the argument is a `symbol`, then it is taken to be the name of a database theory and a text listing of the contents are displayed. Related functions: `settings show-proof-graph`.

show-proof-graph `&optional *proof*` *output.lisp*

Related functions: `show settings`.

show-proofs `&optional proofs` *test.lisp*

Related functions: `test-dtp load-logic-samples`.

show-theory-dag *hierarchy.lisp*

Related functions: `includes unincludes includees decludes included-active-theory-names all-theories`.

test-dtp `&key :reset` *test.lisp*

Related functions: `load-logic-samples show-proofs`.

unincludes `theory-name-1 theory-name-2` *hierarchy.lisp*

Related functions: `includes includees decludes included-active-theory-names show-theory-dag all-theories`.

3.3 Lisp Files

A structured list of the Lisp files in the DTP system is shown in figure 3.2. These files can be found in the logical directory `dtp:code;`

answers.lisp *Knowledge Base*

Successful results of a proof effort. Related files: `literals clauses labels database`.

backtrack.lisp *Inference*

Backjumping,
upon failure in the midst of solving a conjunction. Related files: `misc-inference terms subgoals conjunctions conjunct caching residue ordering prover`.

below.lisp *Graphical Explanations*

Child expansions for existing proof spaces. Related files: `view dotify textify`.

binding-dag.lisp *Unification*

Binding lists. This is a slightly modified version of the file from MVL [Gin]. Related files: `symbols bindings match cnf`.

bindings.lisp *Unification*

Binding lists. This is a slightly modified version of the file from MVL [Gin]. Related files: `symbols binding-dag match cnf`.

caching.lisp *Inference*

Subgoal caching and recursion control. Related files: `misc-inference terms subgoals conjunctions conjunct backtrack residue ordering prover`.

System Definition

`dtp.lisp`

Data Structures

`types.lisp`
`variables.lisp`
`structures.lisp`
`classes.lisp`

Extensions

`internals.lisp`
`defsystem.lisp`

Unification

`symbols.lisp`
`bindings.lisp`
`binding-dag.lisp`
`match.lisp`
`cnf.lisp`

Knowledge Base

`literals.lisp`
`clauses.lisp`
`labels.lisp`
`answers.lisp`
`database.lisp`

Inference

`misc-inference.lisp`
`terms.lisp`
`subgoals.lisp`
`conjunctions.lisp`
`conjunct.lisp`
`backtrack.lisp`
`caching.lisp`
`residue.lisp`
`ordering.lisp`
`prover.lisp`

View

`view.lisp`
`dotify.lisp`
`textify.lisp`
`below.lisp`

Miscellaneous

`hierarchy.lisp`
`output.lisp`
`file.lisp`
`test.lisp`
`epikit-dtp.lisp`

Figure 3.2: File Structure of DTP

classes.lisp *Data Structures*

Class definitions for nodes in DTP proof spaces: subgoals, conjunctions, and conjuncts. Related files: `types variables structures`.

clauses.lisp *Knowledge Base*

Clauses (skolemized logical sentences in conjunctive-normal form). Related files: `literals labels answers database`.

cnf.lisp *Unification*

Database, clausal form. This is a slightly modified version of the file from MVL [Gin]. Related files: `symbols bindings binding-dag match`.

conjunct.lisp *Inference*

Computations for conjunct nodes in the proof space. Related files: `misc-inference terms subgoals conjunctions backtrack caching residue ordering prover`.

conjunctions.lisp *Inference*

Computations for conjunction nodes in the proof space. Related files: `misc-inference terms subgoals conjunct backtrack caching residue ordering prover`.

databases.lisp *Knowledge Base*

Theories and sentences. Related files: `literals clauses labels answers`.

defsystem.lisp *Extensions*

Common Lisp implementation of `defsystem` module. This is a slightly modified version of Mark Kantrowitz's `Portable Mini-DefSystem` (version February 2, 1990). It is available from the CMU common lisp archive. Kantrowitz can be reached at `mkant@GS8.SP.CS.CMU.EDU`. Related files: `internals`.

dotify.lisp *Graphical Explanations*

Proof node to DOT graph conversions. Related files: `view textify below`.

dtp.lisp

Defines the DTP package and system, and loads all the other files.

epikit-dtp.lisp *Miscellaneous*

An example of an interface for making DTP mimic another theorem proving system.[1] User functions: `brf knownp proval remval prologp prologx prologs save drop empty facts content`. Related files: `hierarchy output file test`.

file.lisp *Miscellaneous*

Loading of logical theories from files to lisp data structures. User functions: `dtp-load`. Related files: `hierarchy output test epikit-dtp`.

hierarchy.lisp *Miscellaneous*

Directed acyclic inclusion graph for the theories of logical sentences. Related functions: `all-theories includes unincludes includees decludes included-active-theory-names show-theory-dag`. Related files: `output file test epikit-dtp`.

[1] The other theorem prover in this case is EPIKIT, from Epistemics [GS].

internals.lisp *Extensions*

Simple, generic DTP extensions to Common Lisp. Related functions: `reset-dtp`. Related files: `defsystem`.

labels.lisp *Knowledge Base*

Local labels for logical sentences. Related files: `literals clauses answers database`.

literals.lisp *Knowledge Base*

Logical literals (positive and negative, ground and with variables). Related files: `clauses labels answers database`.

match.lisp *Unification*

Logic variables. This is a slightly modified version of the file from MVL [Gin]. Related files: `symbols bindings binding-dag cnf`.

misc-inference.lisp *Inference*

Generic functions for inference. Related files: `terms subgoals conjunctions conjunct backtrack caching residue ordering prover`.

ordering.lisp *Inference*

Search control (priority queue for the agenda of subgoals). Related files: `misc-inference terms subgoals conjunctions conjunct backtrack caching residue prover`.

output.lisp *Miscellaneous*

General text output to the user. User functions: `settings show show-proof-graph`. Related files: `hierarchy file test epikit-dtp`.

prover.lisp *Inference*

Top-level driver for the theorem prover. User functions: `prove prove-next-answer proof-all-remaining-answers`. Related files: `misc-inference terms subgoals conjunctions conjunct backtrack caching residue ordering`.

residue.lisp *Inference*

Residue (assumable literals) computations. Related files: `misc-inference terms subgoals conjunctions conjunct backtrack caching ordering prover`.

structures.lisp *Data Structures*

Structure objects for DTP proofs. Related files: `types variables classes`.

subgoals.lisp *Inference*

Computations for subgoal nodes in the proof space. Related files: `misc-inference terms conjunctions conjunct backtrack caching residue ordering prover`.

symbols.lisp *Unification*

Logic variables. This is a slightly modified version of the file from MVL [Gin]. Related files: `bindings binding-dag match cnf`.

terms.lisp *Inference*

Procedural attachment for terms, and (hooks for) term inference/rewriting. Related files: `misc-inference subgoals conjunctions conjunct backtrack caching residue ordering prover`.

test.lisp *Miscellaneous*

Runs lisp examples in order to test the entire DTP system. User functions: `load-logic-samples` `test-dtp` `show-proofs`. Related files: `hierarchy` `output` `file` `epikit-dtp`.

textify.lisp *Graphical Explanations*

Proof node to text description conversions. Related files: `view` `dotify` `below`.

types.lisp *Data Structures*

Data types. Related files: `variables` `structures` `classes`.

variables.lisp *Data Structures*

Global parameters, user set-able variables, internal special variables, and initial values. See 3.1 for a complete listing. Related files: `types` `structures` `classes`.

view.lisp *Graphical Explanations*

Top-level drivers for explanation and graph-drawing routines, for both proof spaces and proof answers. User functions: `show` `settings` `show-proof-graph`. Related files: `dotify` `textify` `below`.

Chapter 4

References

[BM] Boyer and Moore. NQTHM: The Boyer-Moore theorem prover. Available online as
`ftp://ftp.cli.com/pub/nqthm/nqthm.tar.Z`
Email contact: `kaufman@cli.com`.

[Fri] Alan M. Frisch. FRAPPS: Framework for Resolution-based Automated Proof Procedures. Available online as
`ftp://a.cs.uiuc.edu/pub/frapps/`
Email contact: `frisch@cs.uiuc.edu`.

[Ged] Donald F. Geddis. DTP: Don's Theorem Prover. Available online as `http://meta.stanford.edu/dtp/`.

[Ged95] Donald F. Geddis. *Caching and First-Order Inference in Model Elimination Theorem Provers*. PhD thesis, Stanford University, Stanford, CA, 1995.

[GF92] Michael R. Genesereth and Richard E. Fikes. Knowledge interchange format: Version 3.0 reference manual. Logic Group Report Logic–92–1, Stanford University, June 1992.

[Gin] Matthew L. Ginsberg. MVL: A multi-valued logic theorem prover. Available online as `ftp://t.uoregon.edu/mvl/`.

[Gin93a] Matthew L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research (JAIR)*, 1:25–46, 1993. Available online as
`ftp://t.uoregon.edu/papers/dynamic.dvi`
JAIR is published in `comp.ai.jair.papers`.

[Gin93b] Matthew L. Ginsberg. *Essentials of Artificial Intelligence*. Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1993.

[GN87] Michael R. Genesereth and Nils J. Nilsson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1987.

[GS] Michael R. Genesereth and Narinder P. Singh. EPIKIT. Available from Epistemics, Inc. Contact the authors at `genesereth@cs.stanford.edu` or `singh@cs.stanford.edu`.

[Kor85] Richard E. Korf. Depth-first iterative deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[Kor92] Richard E. Korf. Linear-space best-first search: Summary of results. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI–92)*, pages 533–538, Menlo Park, California, 1992. AAAI Press.

[Lab] Rewrite Rule Laboratory. RRL: Rewrite Rule Laboratory. Available online as `ftp://herky.cs.uiowa.edu/public/rrl`.

[McC] William W. McCune. OTTER: Organized Techniques for Theorem-proving and Effective Research. Available by anonymous FTP from Argonne National Laboratory. Contact `mccune@mcs.anl.gov` or 708/972–3065.