Notes on MetiTarski Code (to aid future developers)

J Bridge

October 13, 2014

1 Background needed

These notes assume that the reader understands the basic operation of MetiTarski from the published papers (eg Akbarpour and Paulson [1]) and also has an understanding of first order logic resolution theorem provers using the given clause approach. Such topics as splitting with and without backtracking are also covered by published papers [3] [2]. The aim of these notes is to give an overall view of which modules do what at a high level. There is no attempt to describe details at the individual function level or lower.

2 Connection with Metis

Many MetiTarski modules derive from Metis ones, but at the time of writing (September 2014 with MetiTarskic 2.4 vs Metis 2.3 released September 2012) only five modules are unchanged between the two (Lazy.sml, Ordered.sml, Set.sml, Sharing.sml and Heap.sml). Though many other modules are only changed in a minor way.

Modules unique to MetiTarski are : Poly.sml, Polyhash.sml, rat.sml, SMTLIB.sml and all the RCF modules in the RCF subdirectory and the Syntax code.

3 Overview of modules

The modules described here are those actually used in MetiTarski and listed in the file +ld.sml. They are labeled with the ".sml" suffix but most have a corresponding ".sig" signature file. The aim of this overview is to give a developer new to MetiTarski a quick understanding of what each of the different modules do. The key modules are described in more detail later.

3.1 Overall Control Module - metis.sml

Associated module for dealing with some command line options - Options.sml. Options.sml covers the core command line options as existed in the original metis. Later options have been added to metis.sml and if further options need to be added they should be added to metis.sml and not to options.sml.

The MetiTarski version of metis.sml is substantially extended and altered from the Metis version. In addition to code to read problem files and calling the resolution theorem prover in Resolution.sml it also has code to set different options, to automatically select axiom files to include, to repeatedly attempt to find proofs where the prover has given up with one set of parameters/axiom files and also to spawn a separate timing thread to time out on both MetiTarski itself and the corresponding RCF procedure (Qepcad).

Any new options should be added in this module and information transferred to the associated module using the option via reference variables.

3.2 Proof Search Module - Resolution.sml

Resolution.sml is the core module controlling the given clause algorithm for proof search.

For more details see a later section. Associated modules Active, Waiting and SplitStack are described briefly below and in the cases of Active and SplitStack, in more detail later.

3.2.1 Active.sml

Active carries out all inferences on the given clause and adds to the Active set the module contains functions to simplify and factorize.

3.2.2 Waiting.sml

Waiting manages the addition and ordered retrieval of clauses from the Waiting set. This routine also contains the weighting algorithm for clauses to determine which clause is selected to be the next given clause. This should not be confused with the literal weighting algorithm used to determine which literals are eligible for resolution. The literal weighting scheme is contained in KnuthBendixOrder.sml.

Note that splitting with backtracking adds some complexity to the Waiting structure. On backtracking some split levels are deleted and any clauses with labels containing such deleted levels should be deleted.

There is, however, no easy way to delete clauses within the Waiting structure (though an experiment to do so was tried). So instead a lazy approach is used. A list of deleted levels is kept and clauses are only deleted as they reach the point of selection. The function removeIfValid selects the next clause, checks if it is a deleted one and if it is, it then deletes it and selects a replacement clause.

Additionally, backtracking releases clauses in particular the right side of a split. It is easier to add these to Waiting rather than attempt to alter the code to deal with them as soon as they appear. But, in the case of the right split clause it was found to be better if this was given priority and so Waiting was extended to allow a clause to be added to the front of the queue even if its weight would normally place it further back.

3.2.3 SplitStack.sml

SplitStack manages the splitting and backtracking associated with the full splitting option. This code is described in more detail in a separate section.

3.3 Polynomial code - Poly.sml

This code tests if terms are algebraic (suitable for passing to an RCF decision procedure) and if they are then is able to simplify them and place them in canonical form. Poly.sml also contains code to arithmetically simplify terms.

3.4 RCF interface modules

Most of these are in the RCF subdirectory, the top level calls are from the function Thm.decision in Thm.sml. Originally these were just the single file Qepcad.sml that provided a string interface (piped) to Qepcad but now has been greatly extended to not only include files to interface to Mathematica and Z3 but also interval arithmetic and so on (this work mainly done by Grant).

The files fall into one of two classes. There are modules that provide an interface to different external RCF decision procedures in Z3, Mathematica or Qepcad. More recently Grant has generated modules aiming at producing a certified RCF kernel, though initially this is univariate to quote from the change log :

> First round of commits for new certified RCF kernel, including our own implementions (from first principles, and with an eye towards ease of formalisation of the underlying mathematics) of real root isolation based on Sturm's theorem, bivariate resultants via a modular Euclidean method, real algebraic number computations via bivariate resultants, an LCF-style kernel for univariate RCF, and some derived proof procedures.

These additional procedures are committed to the code base but currently are not directly called within MetiTarski unless code is specifically added to do so.

3.5 RCF modules added in March 2014 related to building towards a certified univariate RCF decision procedure

3.5.1 RCF/Resultant.sml

Bivariate Polynomial Resultants via a Modular Euclidean Method.

3.5.2 RCF/Sturm.sml

Sturm Chains and Real Root Isolation.

3.5.3 RCF/RealAlg.sml

Arithmetic in the (Ordered) Field of Real Algebraic Numbers (including interval representation).

3.5.4 RCF/CertRCFk.sml

Proof Procedures for Certified RCF Judgments - An LCF-style kernel.

***** NB This file contains extensive comments on the proof system and related calculus. *****

3.5.5 RCF/CertRCFp.sml

Proof Procedures for Certified RCF Judgments - Derived proof procedures.

***** NB this module contains the main entry point for the univariate RCF decision procedures which produce a full series of inferences that may in future be verified in Isabelle.

3.6 RCF modules relating to interval arithmetic and Groebner basis and polynomial algebra

3.6.1 RCF/Common.sml

Various useful general functions and values used by the RCF routines eg number conversion and character matching.

3.6.2 RCF/Algebra.sml

Routines for basic multivariate polynomial algebra

3.6.3 RCF/Groebner.sml

Groebner basis computation using Buchberger's Algorithm.

3.6.4 RCF/IntvlsFP.sml

Interval arithmetic functions - see comments at start of the file for more detail.

3.6.5 RCF/MTAlgebra.sml

Module connecting MetiTarski formulas to polynomial algebra (see comments in the code for more detail).

3.6.6 RCF/CADProjO.sml

Partial CAD Projection Ordering (i.e. derives a variable ordering for a ground MetiTarski formula based on the Brown heuristic as used in Zongyan's variable ordering for CAD machine learning study.)

3.6.7 RCF/Calibrate.sml

EADM initialisation time calibration for machine learning experiments.

3.6.8 RCF/Nullsatz.sml

A Proof-Producing Exists RCF proof procedure based on the Real Nullstellensatz (including Groebner reduction, in the spirit of Tiwari).

3.7 Modules providing an interface to external RCF decision procedures

3.7.1 RCF/RCF.sml

This is the overall interface to external decision procedures - calls can then be routed to Z3, Mathematica or Qepcad depending on what flags are set.

3.7.2 RCF/SMT.sml

SMT.sml is the interface to Z3 - i.e. writes RCF problems in SMTLib format and sends them to Z3 and reads the responses. Similar to Qepcad.sml which communicates with Qepcad and Mathematica.sml which communicates with Mathematica for RCF decisions.

3.7.3 RCF/Mathematica.sml

Interface to the RCF decision procedure(s) in the Mathematica kernel. Note that MetiTarski can optionally pass expressions containing some special functions as well as purely algebraic expressions.

3.7.4 RCF/Qepcad.sml

Interface to Qepcad for RCF decisions. Character string commands are piped to Qepcad and the resultant string output is analysed.

3.8 MetiTarski specific Data Types

These are mainly inherited from Metis and generally logical (in the literal sense as well as in how they are set out). The exception perhaps is that Thm is at a lower level than Clause (i.e. a clause contains a thm plus further meta information). This can be a little confusing and thms are best looked at as un-numbered clauses, and in these notes I may refer to logical clauses that, in the software, are of type thm. The higher level types carry meta information such as labels used in splitting and backtracking and a history of inferences and parent clauses, this is particularly true of thms.

From lowest level to highest they are:

3.8.1 Name.sml

Names are just defined as strings.

3.8.2 rat.sml

Rat (rational number which is primitive in MetiTarski).

rat.sml contains functions to manipulate rational numbers such as normalising, adding, subtracting, multiplying, dividing, raising to an integer power and so on.

3.8.3 NameArity.sml

String combined with an integer to give function arity.

3.8.4 Term.sml

Terms are variables (defined as a name), a rational number or Rat which is primitive in MetiTarski or a function which is a name*term list (a constant is a function with an empty term list).

3.8.5 Atom.sml

Atoms are predicates but are defined in terms of

relationName which is a Name,

relation which is a relationName * int (the int being the arity)

then an atom is relationName * Term list

Atom.sml contains functions to find names, sub terms, create and destroy equations, check for matches and free variables, perform substitutions and unification and so on.

3.8.6 Literal.sml

Literals are atoms with the addition of a boolean polarity. Most functions in the module are straight forward call throughs to corresponding functions in Atom.sml with due allowance for the sign.

3.8.7 Thm.sml

As well as defining the Thm data type, Thm.sml contains the logical kernel (for first order clausal theorems).

Within Thm.sml the type clause is a simple literal set, this should not be confused with the MetiTarski wide type Clause.clause which is a superset of Thm.thm.

Initially the Thm type consisted simply of clause * (inferenceType * thm list) where the part in parenthesis gives the inference step and the premises (parent clauses) that generated the clause. By following the parent clauses and inference steps back from the empty clause a proof can be reconstructed.

The introduction of splitting with backtracking added the requirement for the thms to have labels which indicate the split levels (in the split tree) on which the the clause is

dependent - the undoing of any such split would then lead to the deletion of the clause as part of backtracking.

In addition, it was found that it was necessary to keep a record of the clause distance in thms so that if they were restored as a result of backtracking they would retain the correct weight - clause distance in MetiTarski is a measure of how far into the search process the clause was generated so that older clauses (with smaller clause distances) are given lower weights than newer clauses (other things being equal).

Finally, a measure of Set Of Support (SOS) was introduced to MetiTarski and an SOS flag also added to the Thm type. The final Thm type is now:

datatype thm = Thm of SOS*clauseDistance * clauseLabel * clause * (inferenceType * thm list)

Note that Thm is a tuple rather than a record so some care is required to note the number and order of elements in the code.

3.8.8 Units.sml

Units.sml is placed above Thm.sml despite the fact that unit clauses are smaller than general clauses. This is because Units.sml contains a data structure for storing thms that are unit with a copy of the single literal of the unit clause being stored as a separate item with the original thm itself.

Units.sml also contains code to reduce the units in the store by matching and resolution.

3.8.9 Clause.sml

The clause type (defined in Clause.sml) adds an ID to a thm and also parameters which define the type of literal ordering to use.

Resolution.sml operates at the clause level (rather than with thms) , as does Waiting and Active.

3.8.10 Formula.sml

Defines a type of first order logic formulas together with routines to extract functions, check equality, find free variables and so on.

3.9 Modules to do with logical inference, rewriting, subsumption, literal ordering for resolution etc.

3.9.1 Subst.sml

Code and data structure (NameMap) for first order logic substitutions.

Includes functions to freshen variables, to find matches and to unify terms as well as more basic operations on the subst data structure.

3.9.2 KnuthBendixOrder.sml

Key module as it determines the weights used for special functions and operators in the KBO of literals to determine allowed resolutions.

KBO is extended with subterm coefficients to give LPO-like properties, see Ludwig and Waldmann [4].

A recent change was to greatly increase the subterm coefficient for unknown (user) functions from 1 to 1000. This means that user functions are given very high priority to be be substituted even when the expression that they are set equal to is quite complex.

3.9.3 Rule.sml

Derived rules for creating first order logic theorems. That is generates thms from (typically) literals based on simple rules such as reflexivity of equality.

3.9.4 Subsume.sml

Datatype and functions for forward subsumption. That is the data structure is designed to hold the clauses that will do the subsuming and is designed for rapid testing for subsumption in that direction (eg by storing units and even empty clauses in separate parts).

Within the context of the given clause algorithm the structure is setup so that the new given clause can be checked against the clauses already in the active set to see if it is subsumed by any of them.

Reverse subsumption, by which a check is done to see if the new given clause subsumes any of the existing clauses in the active set is not catered for.

Experimental code was written to implement reverse subsumption but it was found to very rarely occur in practice so it was not worth creating data structures to implement reverse subsumption efficiently. (The experiments were carried out by implementing it inefficiently and noting how often it was possible.)

3.9.5 Rewrite.sml

Ordered rewriting of first order terms.

3.10 Modules related to file input and output and parsing

Parse code (tables for use with MLLex and MLYacc) are in the Syntax subdirectory.

3.10.1 Syntax/load.sml

The Syntax subdirectory contains the Lex and Yacc generated files as well as the associated grammar. Syntax/load is the main interface module to these.

3.10.2 Parse.sml

Recursive descent parser which predates the parser within the Syntax subdirectory.

3.10.3 Stream.sml

Stream functions and datatype including conversion to and from a text file.

3.10.4 Print.sml

Code for pretty printing.

3.10.5 Tptp.sml

associated module Normalize.sml (puts formula into cnf)

Code and related datatypes to read files in TPTP format.

3.10.6 Proof.sml

Code to reconstruct the proof from thms which contain both parent clauses (premises) and the inference that generated them.

Note that for proofs where splitting with backtracking has been used much of the proof reconstruction is done as part of the process of restoring hidden literals and the code for this can be found (with comments) in SplitStack.sml.

3.10.7 SMTLIB.sml

Code to read (parse) and write SMTLIB format files. The output is a Tptp.problem.

3.11 Low level modules associated with generic data structures etc.

3.11.1 useful.sml

various useful functions for inclusion in other modules

3.11.2 Lazy.sml

code for lazy evaluation which is only used by Stream.sml I think.

3.11.3 Random.sml

Pseudo random number generator (with some real randomness incorporated via timing)

3.11.4 Sharing.sml

functions designed to preserve the sharing of ML variables, only the Sharing.map function is used as far as I can ascertain.

3.11.5 Portable.sig, PortablePolyml.sml, PortableSmlNJ.sml

modules to allow the portable comparison of pointers and other compiler dependent operations.

3.12 Generic data structures and associated instantiations

3.12.1 Polyhash.sml

Hash table code - not related to the polynomial routines found in Poly.sml other than it was introduced to enable the storage of RCF decisions for sets of polynomials so that repeat decisions can be determined from the hash table rather than from a repeated call to the slow RCF decision procedure.

3.12.2 Map.sml

Finite maps implemented with randomly balanced trees.

3.12.3 Set.sml

Finite sets implemented with randomly balanced trees.

3.12.4 Ordered.sml

integer or lexicographical compares defined

3.12.5 KeyMap.sml

Finite maps with a fixed key type.

3.12.6 ElementSet.sml

Finite sets with a fixed element type.

3.12.7 TermNet.sml

Matching and unification for sets of first order logic terms. The functions return over-approximations for matching and unification so post filtering is required.

3.12.8 AtomNet.sml

Matching and unification for sets of first order logic atoms - makes use of TermNet.sml.

3.12.9 LiteralNet.sml

Matching and unification for sets of first order logic literals - makes use of AtomNet.sml

3.13 Models

Models appear in MetiTarski in two contexts. The original Metis code allows models of finite domains and the code is contained in model.sml. This feature is not used in MetiTarski as it was found not to work well but it is still part of the code.

The second context in which models are used is in the RCF code. Counter examples may be saved as models for reuse to speed up the RCF decision process.

4 Key modules in more detail

4.1 Resolution.sml & Active.sml

These two modules are considered together as they both combine to implement the given clause algorithm in the proof search.

NB MetiTarski uses the Discount loop rather than the Otter loop. The difference between the two is given in the following quote from the Prover 9 manual:

> There are two common versions of the given clause algorithm that differ in how and when simplification (i.e., rewriting) occurs. In the Otter loop, which Prover9 uses, clauses in the sos list can simplify new clauses, and new simplifiers are applied immediately to all clauses, including sos clauses. In the Discount loop, clauses in the sos list (also called the passive list) cannot simplify or be simplified until they are selected as given clauses. The tradeoff between the two versions is straightforward — the Otter loop spends much more time simplifying with the possible benefit of making an important simplification sooner.

In MetiTarski the number of simplifications is further reduced as clauses already in the Active set are used to simplify the given clause but the given clause is not used to simplify the clauses already in Active at least as far as subsumption is concerned. An experiment was performed in which such backward subsumption was implemented (albeit using a non-optimised structure and slow code) and it was found that it almost never occurred even given splitting which may lead to the given clause being lighter than earlier clauses. It was therefore considered not worth introducing complex new data structures to implement it as a feature of MetiTarski.

4.1.1 Process steps

The main loop is in function iterate.

The first step is to extract a new "given clause" from the Waiting clause set using the function Waiting.removeIfValid, but this is only done if the number of previously extracted clauses that are not in the SOS (set of support which is those clauses that are part of the conjecture or inferred from such clauses) does not exceed a preset limit.

Assuming that a new clause is available (see next section for what happens if there is no more clauses in Waiting):

If the new clause is algebraic (and splitting with backtracking is being used), check for consistency with existing algebraic clauses. If it is inconsistent then it is replaced by an empty clause. NB that with splitting with backtracking, an empty clause is not always the end of the overall proof search but just an indication that a branch has been closed and the other branch should be explored. The difference between the two cases depends on the clause label attached to the empty clause.

Note that in the case of the new clause being made empty by being inconsistent with the current set of algebraic clauses, then the original clause needs to be added to Waiting. The reason for this is that the empty clause will cause backtracking. The backtracking may affect the algebraic clauses such that the original inconsistency is removed. This will lead to the original clause being restored and the simplest mechanism is to just add it to Waiting. In my notes of 27/4/11 I argued that the "subsuming" clause would be the existing empty clause which is then removed by backtracking so that adding it to the deleted clauses would just cause instant restoration anyway.

The case of the clause being the empty clause leading to backtracking is covered in a following section.

If the clause is not empty it is then checked to see if it is a new clause or simply a logical consequence of existing algebraic clauses.

Assuming the clause is a new one, it is then checked to see if it is an algebraic clause and added to the set (list) of algebraic clauses if it is.

The current clause distance (a measure of how far the proof search has progressed) is then set to the value extracted from the clause. Prior to backtracking being implemented the clause distance was simply incremented with each new given clause but with backtracking clause distance is attached to the clause itself so that right split clauses are not unfairly penalised relative to left split clauses and similarly restored clauses have a correct clause distance.

The implementation of the next steps is divided between Resolution.sml and Active.sml differently depending on whether pseudo-splitting or splitting with backtracking is being used.

The steps are

1. simplify
2. split (if possible)
3. add clause to the active set
4. rename the variables in the clause to produce a separate copy
5. carry out deduction (i.e. all possible inferences with the given clause and the active set)
6. factor the new clauses
7. apply arithmetic simplification to the resultant new clauses

With pseudo-splitting all these stages are contained in the single function Active.Add. With splitting with backtracking the function needed to be split as the splitting stage is carried out in Resolution.sml rather than Active.sml (it was simpler to implement this way).

So for the latter case stage 1.) is in function Active.simplify and stages 3.) to 7.) are in function Active.addAndFactor.

Following the addition of the clause to the Active set and the corresponding generation of new clauses from inferences, the new clauses are further simplified (if possible) by using RCF decision procedures to determine if literals may be deleted given the current set of algebraic clauses as additional constraints (see Akbarpour and Paulson for the details [1]).

The penultimate step is to assign a new clause distance by adding to the existing distance a number dependent on the number of new clauses - the increment differs depending on whether pseudo-splitting or splitting with backtracking is being used. The reason for the difference is that in experiments a simpler expression was found to be just as good and implemented in the newer splitting with backtracking but for backward compatibility the older expression was retained for the pseudo-splitting case.

Finally the new clauses are added to the Waiting set.

4.1.2 Backtracking on finding the empty clause

On finding the empty clause if the level is greater than zero (i.e. some backtracking is required) then SplitStack.backtrack is called. The result of such backtracking is either a top level empty clause (i.e. the overall proof has been found) or a right branch which needs to be explored.

The right branch (if there is one) is added to Waiting but it is added to the front of the queue rather than placed according to its weight using the function Waiting.addToFrontOfQueue.

Clauses restored as the result of backtracking are added to Waiting in the normal way.

4.1.3 What happens if there are no further clauses in the Waiting set

Given the weight limit on clauses kept in Waiting, the absence of clauses does not imply saturation so the normal action is to return in the state "Gave Up". But MetiTarski allows some iterative deepening of the search if the correct flags are set. Within Resolution.sml if there is an upper limit on the number of symbols allowed in algebraic clauses then on reaching an empty Waiting set the limit is set to a high maximum and the search restarted.

Note also that there is a degree of iterative deepening available at the axiom level with the autoInclude option. This is carried out within the routine metis.sml rather than in the proof loop in Resolution.sml.

4.2 SplitStack.sml - Splitting with Backtracking

4.2.1 Background

The splitting method and nomenclature used is based on the Fietzke and Weidenbach [3] but there are some differences which are covered in our paper (Bridge and Paulson [2]) and in these notes. It is expected that the reader of these notes has already read these two papers. The main reason for changing the exact structure used by Fietzke and Weidenbach is that their design decisions were constrained by the existing SPASS structure and not having these constraints in MetiTarski allowed us to separate deleted clause lists from the split stack structure for example.

Note that splitting in the MetiTarski case is simplified by the clause being ground and the left split being a single literal, but the code has been written to be more general by not assuming that this is the case. In particular the left split is checked for being ground before being negated and the negation allows for it to contain more than a single literal (the negation of multiple literals results in multiple clauses).

4.2.2 Clause Labels and Hidden Literals

For splitting with back tracking the clause labels encapsulate a splitting tree or the history of splits that a particular clause arises from. These labels are part of the thm structure and should not be confused with clause "labels" that are used in the pseudo splitting (without backtracking) where the labels are additional literals.

One way of viewing clause labels is as encoding literals which should be part of the clause but are hidden as they are assumed to be false (splitting is basically making the assumption that certain literals in a clause are false and then returning to this assumption when the remaining literals have been deleted).

This idea of hidden literals is not relevant to the proof search itself, but it does allow the generation of a non-splitting version of the proof once it has been found. For example, consider a clause which prior to splitting is $A \vee B \vee C \vee D$. The left split is just A with an associated label. The label can be considered to embody the "hidden" literals $B \vee C \vee D$. Any inference step which has A or a derived clause as a premise is dependent on $B \vee C \vee D$ being false (the assumption of the split) so the resultant generated clause (which inherits the label) will also be dependent on $B \vee C \vee D$ being false.

If the empty clause $\Box$ is reached then this is equivalent to the reduced clause $B \vee C \vee D$ which is the right branch which must then be explored.

(Note that with many splits the labels and hidden literals become compounded but I've ignored this for simplicity.)

When reproducing a proof, rather than having splitting as an inference step instead the hidden literals can be reinstated.

A further simplification can be made - given that the left split is typically a single literal in MetiTarski - and that is to derive the left split A from an assumption followed by a left split:

assume $A \vee \neg A$

then the hidden literals (including any from earlier splits) can be replaced by the single literal $\neg A$.

The right hand side of the branch is then recovered by resolution of $\neg A$ with the original clause $A \vee B \vee C \vee D$.

A big proviso or course is that the clause is ground so that there are no variables in common between the different splits. This is the case in MetiTarski.

4.2.3 The Split Stack

As in Fietzke and Weidenbach's [3] work with SPASS a stack is used to keep track of the splits made and to allow the unravelling of the splits when backtracking occurs. The stack used in MetiTarski is simplified relative to the one used in SPASS by separating deleted clauses into a separate list. Searching a list is much simpler than checking through a mixed-use stack and it is likely that the SPASS approach was used for historical reasons to fit in with an existing structure.

Clause "deletions" arise mainly from two different types of inferences. Straight forward deletion arises when a clause is subsumed by another clause. A more indirect method is when a clause is simplified and replaced by its simplified form. The simplified clause generally does subsume the unsimplified clause but it is not necessary to formally check this for the system to work. Initially the deleted clause list contained both the subsuming clause and the deleted clause but following a suggestion from one of the reviewers of an early version of our paper, the subsuming clause was replaced by just its label. The clause label contains all the information pertaining to which splits the deletion process is dependent on and so if any of the splits in the label have been backtracked over then the corresponding deleted clause is restored.

A complication with the data structures inherited from Metis and backtracking occurs when clauses are deleted from the active set. Some care needs to be taken when deleting clauses from the subsumption data structure for example to ensure all associated entries are removed and the structure retains its consistency.

A form of simplification peculiar to MetiTarski is literal deletion arising from RCF decisions which also take into account a list of existing algebraic clauses. In this case the label of the simplified clause will be an amalgamation of the labels of all the algebraic clauses even if some of such clauses are not strictly needed. One experiment that was tried in the earlier versions of the backtracking version of MetiTarski was to keep separate lists of algebraic clauses, one for top level clauses that don't rely on any splits and the other for all algebraic clauses. It turned out that the added complication was not justified by any noticeable improvement in results.

5 Rewrites and Factoring in Metis

The following is from an e-mail I wrote on 9/3/12. It highlights some of the issues involved in combining the "factoring" operation with other inference steps especially when backtracking is involved. In particular the simplification of initial axioms by

factoring (and the associated inferences done with the factors) can lead to problems with some proofs so this is not done in MetiTarski. I first looked into this as the result of some odd behaviour associated with paramodulation/demodulation in MetiTarski.

In Metis factoring (which is applied to a list of clauses) involves:

i.) for each clause finding all possible factors (pre and post factoring simplification is also carried out, in particular subsumption).

ii) new clauses are (temporarily) added to the existing subsume structure and subsumption applied where possible (in particular at least one factor will subsume the parent clause). As an aside this made life complicated from the backtracking point of view where I had to transfer a list of deleted clauses when the temporary subsumption structure was discarded.

iii) Where possible factors are added to the rewrite structure and the unit clauses in the active set (this is possibly a bit dodgy in the light of backtracking but for other reasons the code does not add rewrites which are not top-level so I think it is ok, though possibly I should do this for unit clauses as well).

iv) all clauses that might be simplified (by rewrites) are then extracted from the active set and are in turn pass to the factor routine.

v) NB as part of the process of extracting rewriteable clauses (step iv) above, the rewrite structure within active is maximally reduced by applying rewrites between themselves - it is this which has had the odd side effects with the equality axioms we've been looking at.

Factoring is applied to all clauses at different points.

a.) for axiom and conjecture clauses it is applied when the active structure is set up so this means even though the clauses themselves are added to waiting they have a ghostly presence within the rewrite/unit clause structures in the active structure.

b.) factoring is applied to simplified clauses and newly deduced clauses just before they are added to waiting.

c.) a clause being added to active that is not simplified is NOT factored - this is because it should either have been factored when it was deduced or it should have been factored as one of the axiom of conjecture clauses.

So when I removed the factoring of axioms this meant that such clauses (the ones that couldn't be simplified) were never factored which had odd effects for the (a=6) type axioms.

6 Subsumption

When checking whether any clause in Subsume.subsume subsumes a given clause cl (a.k.a. forward subsumption) the following algorithm is used:

1. If there's an empty clause in Subsume.subsume, then return it.

2. Check whether any of the unit clauses in Subsume.subsume matches a literal of cl.

3. The rest of the clauses in Subsume.subsume are stored with two of their literals treated specially: one goes into a set fstLits; and another goes into a set sndLits.

3a. Let C1 be the set of clauses in Subsume.subsume containing a literal in fstLits that matches a literal in cl.

3b. Let C2 be the set of clauses in Subsume.subsume containing a literal in sndLits that matches a literal in cl.

3c. Let C = C1 intersect C2 be the set of candidate subsumption clauses in Subsume.subsume.

3d. For each clause in C, perform a full subsumption check with cl. Return the first clause that succeeds.

Note that the subsumption structure allows empty clauses which will subsume any clause. In the normal operation of Metis/MetiTarski the appearance of an empty clause would signal a successful proof. However, the forward subsumption data structure is written to be generic and to store empty clauses that are added and return them as subsumption matches.

7 Termnets

The Metis code in Termnets.sml was developed independently by Joe Hurd but similar ideas are embodied in McCune [5].

McCune's Jump Lists always include forward pointers, and plain discrimination trees never do. Metis's term-nets include forward pointers only for Single constructors (see below).

The Metis term-net works by flattening each term to a list (using a pre-order traversal) and then combining terms with a common list prefix to form a tree with Multiple nodes at the branch points and Result nodes at the leaves. The initial implementation had just these constructors, but the common case is that when traversing the tree from root to leaf there will be some initial Multiple constructors with more than one child, then a long sequence of Multiple constructors with just one child branch eventually leading to a Result. The Single constructor is an optimization for the long sequence portion of the tree by storing whole subterms instead of flattening them into lists. As a side-benefit the Single constructor allows you to easily skip subterms a la jump lists.

Though the Single constructor was created with the purpose of optimizing long Single sequences leading to Results, they can appear in other places too. For example, consider adding the two terms

f(t,u)

f(t,v)

to an empty term-net. The resulting term-net will look something like

Multiple (NONE, f/2 |–>Single (t, Multiple (...)))

where the Single is used to indicate that at the subterm position "first argument of f" it has only ever seen one term "t". The second argument of the Single constructor is a choice point using Multiple, because there are two terms "u" and "v" to be discriminated at the next subterm position "second argument of f".

References

[1] Behzad Akbarpour and Lawrence C. Paulson. Metitarski: An automatic prover for the elementary functions. In *Proceedings of the 9th AISC International Conference, the 15th Calculemas Symposium, and the 7th International MKM Conference on Intelligent Computer Mathematics*, pages 217–231, Berlin, Heidelberg, 2008. Springer-Verlag.

[2] James P. Bridge and Lawrence Charles Paulson. Case splitting in an automatic theorem prover for real-valued special functions. *Journal of Automated Reasoning*, 50(1):99–117, 2013.

[3] Arnaud Fietzke and Christoph Weidenbach. Labelled splitting. In *Proceedings of the 4th International Joint Conference on Automated Reasoning*, IJCAR '08, pages 459–474, Berlin, Heidelberg, 2008. Springer-Verlag.

[4] Michel Ludwig and Uwe Waldmann. An extension of the knuth-bendix ordering with lpo-like properties. In Nachum Dershowitz and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 4790 of *Lecture Notes in Computer Science*, pages 348–362. Springer Berlin Heidelberg, 2007.

[5] William McCune. Experiments with discrimination-tree indexing and path indexing for term retrieval. *JOURNAL OF AUTOMATED REASONING*, 9:9–2, 1990.