PSYCHE

a proof-search engine based on sequent calculus with an LCF-style architecture

Stéphane Graham-Lengrand

PSI project, CNRS-Ecole Polytechnique-INRIA,

Tableaux'13, 19th September 2013

PSYCHE **in a nutshell**

PSYCHE is a modular proof-search engine

designed as a platform for automated or interactive theorem proving

- kernel/plugin architecture with LCF-style interface & guarantees
- implementing bottom-up proof-search in Sequent Calculus
- + ability to call decision procedures
- can produce proof objects (output in e.g. LaTeX, though quickly too big)

Early days: version 1.5 released (April), version 2.0 in progress

Contents

I. Motivation

Ménagerie

Tools concerned with theorem proving (in a large sense):

1. Automated Theorem Provers
2. SAT/SMT-solvers
3. Proof assistants
4. “Logic programming” languages
5. . . .

A lot of research on making them collaborate:

(1+2), (1+3), (2+3),. . .

Central to collaborations: the question of trust

Active research on proof formats and proof exchange (e.g. PxTP workshop)

Possibly use backend proof-checker.

Different research efforts in that direction

- Translating to Coq, proofs from other provers
- Dedukti, based on Deduction Modulo (Dowek et al.)
- Miller's ProofCert project @ Parsifal

(Not concerned with the way external tools have found their proofs)

Or: **LCF** (as in e.g. Isabelle) where every implementation of technique separates:

- the code implementing the actual reasoning steps (the same for everyone)
  concerns correctness of answer
- the code implementing strategies concerns efficiency of producing answer

so that "answers" are correct-by-construction (no proof-checker needed)

LCF style

Kernel knows of private type `thm` for theorems

offers API so proof-construction becomes programmable outside kernel

producing inhabitants of `thm`

$\Longrightarrow$ output trusted as correct if kernel is trusted

LCF highly programmable, but kernel is of little help for the proof-search per se

Primitives are for proof reconstruction rather than proof-search.

Besides internal tableau implementation, Isabelle can use Metis+Sledgehammer to delegate the search to on-the-shelf black boxes (SMT-solvers, ATPs).

PSYCHE experiments a new version of LCF

where the *kernel* performs some actual proof-search "*à la* Prolog",

while leaving heuristics to be programmed as *plugins*

Experimented by implementing **DPLL(T)** as plugin

PSYCHE = Proof-Search factorY for Collaborative HEuristics

II. PSYCHE's architecture

A stupid metaphore

Interaction between a kernel, a theory and a plugin

Theory = land/terrain

Kernel = road network + a car moving on it

Plugin = driver in the car

Common objective: reach a destination

Correctness:

interaction between Kernel and Plugin is organised so that the car stays on the road

cannot claim the destination is reached if it isn't

In other words: trust the car for correctness, hope driver is efficient at driving it

Driver gets into unfamiliar neighbourhood? Change driver!

More seriously

Kernel knows search-space, which portion has been explored, which remains to be (takes branching and backtracking into account)

Plugin drives kernel through search-space (which branch explore first? which depth?)

Kernel says when a proof has been found, or no proof exists — Not the plugin

Safety of output

How? As in LCF-style, a private type (known only to kernel) is used

Given logical rule

$$\frac{\text{prem}_1 \quad \ldots \quad \text{prem}_n}{\text{conc}}\ \text{name}$$

LCF-style kernel offers API primitive `name:` **thm** `->` $\cdots$ `->` **thm** `->` **thm**

with `thm` private type

In PSYCHE, rule wrapped in unique API primitive:

`machine:` **statement** `->` **output**

such that `machine(conc)` triggers recursive calls

`machine(prem_1)`,..., `machine(prem_n)`

How it is structured

Top-level

- organises parsing of input
- initialises sequent to prove
- calls

  `Plugin.solve(Kernel.machine(Parser.parse input))`

For plugin, output type of `Plugin.solve` is abstract:

it cannot construct a value of that type,

can only pass on a value provided by (`Kernel.machine`)

= plugin cannot cheat

= no need to understand or certify plugin's code to have a guarantee about the output

Kernel = slot machine

Plugin computes after kernel? not quite

```
type output = Final of answer | Fake of coin -> output
```

kernel's machine outputs

- either final answer provable or not provable
- or "fake" output (= unfinished computation):

  for computation to continue, plugin "inserts another coin in the slot machine";

  depending on coin, proof-search will resume in a certain way.

In brief: Kernel performs proof-search while no decision needs to be made

(on which backtrack may later be needed)

stops and asks further instructions from plugin when decision needs to be made.

Objective: hit jackpot with kernel outputting value `Final(...)`

```
type answer = Provable of statement*proof | NotProvable of statement
```

answer is private

Contents of /src

/src/.	top-level (156 lines)
/src/run_tools	IO (192 lines)
/src/parsers	(DIMACS, SMTLib2) (428 lines)
/src/lib	common library (518 lines)
/src/kernel	kernel files (586 lines)
/src/generic-plugins/Common	commons files for plugins (219 lines)
/src/generic-plugins/DPLL_WL	plugin DPLL_WL (361 lines)
/src/generic-plugins/...	future plugins
/src/theories/Empty	propositional logic (231 lines)
/src/theories/LRA	linear rational arithmetic (997 lines)
/src/theories/...	future theories

III. PSYCHE's kernel

Generalities

The kernel is an implementation of a focused sequent calculus, which provides a "natural generalisation" of logic programming beyond Horn clauses / HH formulae

Logic of PSYCHE 1.5: polarised quantifier-free classical logic modulo theories

Why polarised?

- inference rules = basic reasoning steps with which proving techniques (i.e. the plugins) are implemented
- different inference rules for $\wedge^+$ and $\wedge^-$, for $\vee^+$ and $\vee^-$

  = more proof-search primitives offered to implement plugins
- polarisation identifies:

  reasoning steps that are w.l.o.g (invertible inference rules)

  from reasoning steps creating backtrack point (non-invertible inference rules)

Logical system

$$A, B, \ldots ::= \; l \mid A \wedge^{+} B \mid A \vee^{+} B \mid A \wedge^{-} B \mid A \vee^{-} B$$

involutive negation on literals l, extended to all formulae

Intuition:

negatives have invertible introduction rules

positives are their negations

Literals are not a priori polarised proof-search will polarise them on the fly

Focusing is the ability to recursively chain decomposition of positives without loosing completeness:

Just after decomposing $(A_1 \vee^{+} A_2) \vee^{+} A_3$ by going for the left, we can assume wlog that we can directly go for A_1 or A_2 instead of working on another formula (we don't risk loosing provability)

Inference rules (similar to Liang-Miller's LKF)

$\mathcal{P}$: set of literals declared to be positive (negations are negative)

Γ: (multi)set of positive literals, Δ: (multi)set of positive formulae

Synchronous phase

$$\frac{\Gamma \vdash_{\mathcal{P}} [A]\Delta \qquad \Gamma \vdash_{\mathcal{P}} [B]\Delta}{\Gamma \vdash_{\mathcal{P}} [A \wedge^{+} B]\Delta} \qquad \frac{\Gamma \vdash_{\mathcal{P}} [A_i]\Delta}{\Gamma \vdash_{\mathcal{P}} [A_1 \vee^{+} A_2]\Delta}$$

$$\frac{}{\Gamma, p \vdash_{\mathcal{P},p} [p]\Delta} \qquad \frac{\Gamma \vdash_{\mathcal{P}} N \mid \Delta}{\Gamma \vdash_{\mathcal{P}} [N]\Delta}\ N \text{ not positive}$$

Asynchronous phase

$$\frac{\Gamma \vdash_{\mathcal{P}} A, \Pi \mid \Delta \qquad \Gamma \vdash_{\mathcal{P}} B, \Pi \mid \Delta}{\Gamma \vdash_{\mathcal{P}} A \wedge^{-} B, \Pi \mid \Delta} \qquad \frac{\Gamma \vdash_{\mathcal{P}} A_1, A_2, \Pi \mid \Delta}{\Gamma \vdash_{\mathcal{P}} A_1 \vee^{-} A_2, \Pi \mid \Delta}$$

$$\frac{\Gamma \vdash_{\mathcal{P}} \Pi \mid \Delta, P}{\Gamma \vdash_{\mathcal{P}} P, \Pi \mid \Delta}\ P \text{ positive} \qquad \frac{\Gamma, l^{\perp} \vdash_{\mathcal{P};l^{\perp}} \Pi \mid \Delta}{\Gamma \vdash_{\mathcal{P}} l, \Pi \mid \Delta}\ l \text{ not positive}$$

Structural rule

$$\frac{\Gamma \vdash_{\mathcal{P}} [P]\Delta, P}{\Gamma \vdash_{\mathcal{P}} \ \mid \Delta, P}$$

Properties

Cuts are admissible, such as:

$$\frac{\Gamma \vdash_{\mathcal{P}} A \mid \Delta \quad \Gamma \vdash_{\mathcal{P}} A^{\perp} \mid \Delta}{\Gamma \vdash_{\mathcal{P}} \ \mid \Delta}$$

System is sound and complete for pure propositional logic,

no matter the polarities of connectives and literals

(these only affect shapes of proofs / algorithmics of proof-search)

Is extended in PSYCHE for quantifier-free logic **modulo theories**:

sound and complete (provided some condition on the polarity of literals)

Can be extended to first-order logic (∀ is negative, ∃ is positive)

Division of labour

Kernel knows the rules

applies asynchronous rules automatically

until hits point with choice and potential backtrack

At each of those points, plugin instructs kernel how to perform synchronous phase

Kernel records alternatives when plugin makes choice

organises backtracking

realises by itself when backtrack points are exhausted and no proof has been found

IV. PSYCHE's plugins: My first SAT-solver

Motivation. . .

. . . was to make different techniques available on the same platform

Challenge:

understand each technique as bottom-up proof-search in focused sequent calculus

Each technique / each combination of techniques,

is to be implemented as an OCaml module of type

```
module type PluginType = sig
  ...
  solve:  output->answer
end
```

PSYCHE works with any module of that type

Today

We know how to do

- analytic tableaux (closest to sequent calculus)
- clause tableaux
- ProLog proof-search
- Resolution
- DPLL(T)
- human user

In PSYCHE 1.5 we have implemented

- DPLL(T)

We investigate how to do

- controlled instantiation using triggers
- specific treatment of equality

DPLL

- **Decide**:

  $\Gamma \| \phi \Rightarrow \Gamma, l^d \| \phi$ where $l \notin \Gamma, l^\perp \notin \Gamma, l \in \text{lit}(\phi)$

- **Fail**:

  $\Gamma \| \phi, C \Rightarrow$ UNSAT if $\Gamma \models \neg C$ and there is no decision literal in Γ

- **Backtrack**:

  $\Gamma_1, l^d, \Gamma_2 \| \phi, C \Rightarrow \Gamma_1, l^\perp \| \phi, C$ if $\Gamma_1, l, \Gamma_2 \models \neg C$ and no decision literal is in Γ_2

- **Unit propagation**:

  $\Gamma \| \phi, C \vee l \Rightarrow \Gamma, l \| \phi, C \vee l$ where $\Gamma \models \neg C, l \notin \Gamma, l^\perp \notin \Gamma$

lit(ϕ) denotes the set of literals that appear / whose negation appear in ϕ

How it is represented in sequent calculus

A clause $C = l_1 \vee \ldots \vee l_p$ is represented in sequent calculus by $l_1 \vee^- \ldots \vee^- l_p$, so $C^\perp = {l_1}^\perp \wedge^+ \ldots \wedge^+ {l_p}^\perp$

DPLL starts with a state $\emptyset \| C_1, \ldots, C_n$

in sequent calculus we try to prove $\vdash \ | \ {C_1}^\perp, \ldots, {C_n}^\perp$

DPLL finishes on UNSAT $\Leftrightarrow$ proof constructed in sequent calculus

DPLL finishes on model $\Leftrightarrow$ no proof exists in sequent calculus

Intermediary states $\|C_1, \ldots, C_n \quad \Longrightarrow^* \quad \Gamma \| C_1, \ldots, C_n$ of DPLL:

in sequent calculus

- we have constructed a partial proof-tree of $\vdash \ | \ {C_1}^\perp, \ldots, {C_n}^\perp$
- we are left to prove $\Gamma \vdash_\Gamma \ | \ {C_1}^\perp, \ldots, {C_n}^\perp$
- each decision literal in Γ corresponds to a branch of the proof-tree being constructed, that is still open

How DPLL is simulated in sequent calculus

Fail using clause C	$\Leftrightarrow$	Focus on $C^\perp$
Backtrack using clause C	$\Leftrightarrow$	Focus on $C^\perp$
Unit propagate using clause C	$\Leftrightarrow$	Focus on $C^\perp$
Decide	$\Leftrightarrow$	Cut-rule (analytic cases!)

What about more sophisticated features of DPLL?

Backjump and Learn cut a lot of branches

Forget and Restart can speed up the process as well

Restart in PSYCHE:

plugin can keep track of 1st plugin-kernel interaction and resume there

Backjump and Learn can be simulated as proof-search by extending several branches of incomplete proof with the same steps.

To do this efficiently in PSYCHE:

Memoisation of the proof-search function

V. Last few things before demo

Theories

Again, Theory = any OCaml module of type

```
module type TheoryType = sig
  ...
  consistency:  literals set -> (literals set) option
end
```

Currently implemented as such a module:

- empty theory (propositional logic)
- LRA
- congruence closure

DEMO

VI. Conclusion

Conclusion

Current plugins and decision procedures are illustrative toys

DPLL plugin very basic (although already implements watched literals)

LRA decision procedure “quickly written”, not incremental

PSYCHE is a platform where people knowing good and efficient techniques should be able to program them

Further work (nothing surprising):

- improve current decision procedures and add new ones
- add new techniques as plugins (e.g. user-interactive)
- proof-terms and classical program extraction

More excitingly, version 2.0 will

- gain level of abstraction by having logic as a parameter (intuitionistic, classical,. . .)
- “handle” quantifiers, capturing triggers-based instantiation mechanisms of SMT, and propagating semantical constraints on meta-variables through branches (D. Rouhling & SGL axiomatised the specs of decision procedures for this to work)

Thank you!

www.lix.polytechnique.fr/~lengrand/Psyche