PSYCHE

a proof-search engine based on sequent calculus

with an LCF-style architecture

Stéphane Graham-Lengrand

CNRS - Ecole Polytechnique - INRIA,

Gallium, 19th June 2014

PSYCHE in a nutshell

PSYCHE is a modular proof-search engine written in OCaml (4.00 at the moment)

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 1.6 released in May (5547 l.o.c.), version 2.0 in progress

Contents

I. **Motivation**

II. PSYCHE**'s architecture**

III. PSYCHE**'s kernel**

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

V. **Last few things before demo**

VI. **Current work on quantifiers**

VII. **Conclusion**

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 @ Deducteam
- 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

(constructors of `thm` not known outside kernel)

offers API so proof-construction becomes programmable outside kernel

producing inhabitants of `thm`

$\Longrightarrow$ inhabitants of `thm` are trusted as proved theorems if kernel is trusted

(regardless of the rest of the code)

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

Proof-search in LCF

LCF API's primitives are for proof reconstruction rather than proof-search:

Given logical rule

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

kernel offers *top-down* API primitive $\texttt{name:}\quad \textbf{thm} \texttt{ -> } \cdots \texttt{ -> } \textbf{thm} \texttt{ -> } \textbf{thm}$

Proof-search usually uses above rule *bottom-up*, as in tableaux or Prolog

(but not the inverse method)

Bottom-up proof-search would have to be programmed outside kernel in

Continuation-Passing-Style

Kernel does not organise exploration of search-space, especially backtracking

LCF architecture guarantees *correctness* of answer, not *completeness*

. . . experiments a new version of LCF

where the *kernel* performs some actual proof-search “*à la* Prolog”

However, efficient proof-search requires the use of heuristics

whose code should certainly not be in kernel

it’d be nice if each call to kernel’s primitives terminated

Proposed solution: split the code of proof-search between

1. systematic steps that can be performed wlog and without intelligence
2. steps that require smart choices to be made by heuristics

Exercise: in Prolog? More generally: **focusing** provides such a division of labour

1. to be performed by kernel
2. to be programmed as *plugins*

Different division of labour than in traditional LCF

Different approaches

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

In PSYCHE we open black boxes, reprogram their algorithms directly in LCF style

Black box approach requires

- feeding black box with input that it can entirely treat

  (usually, abstraction of current proof state)

- waiting until the call finishes (no progress is made until then)

...whereas PSYCHE could run a technique up to a point,

then possibly change technique according to the shape of new proof state

$\Longrightarrow$ new possibilities of technique combination

PSYCHE = Proof-Search factorY for Collaborative HEuristics

So far, we experimented PSYCHE by implementing **DPLL(T)** as plugin

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

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

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 call

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

For plugin, output type of `Plugin.solve` (called **answer**) 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** `| Temp of` **info**`*(`**coin**`->`**output**`)`

kernel's machine outputs

- either final answer provable or not provable
- or "temp" 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 (237 lines)
`/src/run_tools`	IO (193 lines)
`/src/parsers`	(DIMACS, SMTLib2) (367 lines)
`/src/lib`	common library (518 lines)
`/src/kernel`	kernel files (**614** lines)
`/src/generic-plugins/Common`	commons files for plugins (312 lines)
`/src/generic-plugins/DPLL_WL`	plugin DPLL_WL (406 lines)
`/src/generic-plugins/...`	
`/src/theories/Empty`	propositional logic (234 lines)
`/src/theories/LRA`	linear rational arithmetic (1000 lines)
`/src/theories/CC`	congruence closure (1074 lines)
`/src/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 1.6: 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 1.6 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

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

Restart in PSYCHE:

plugin keeps track of 1st plugin-kernel interaction and resumes there

implemented in PSYCHE 1.6

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

Implementation choices

In OCaml

3 data structures on which Kernel+Plugin have to agree

- Data structure for the implementation of formulae
- Data structure for the implementation of sets of formulae
- Data structure for the implementation of sets of literals

“have to agree” = currently provided by plugin

Correctness of data structures have to be checked/assumed

to guarantee correctness of the whole thing

Implementation issues

Most data structures are using hash consing techniques:

a given structure can only be stored in memory once

Memoisation table is implemented using Patricia tries

(persistent yet efficient representation of sets and maps

unique representation of a given set or map

they are hash-consed)

DEMO

VI. Current work on quantifiers

Quantifiers

$$\frac{\Gamma \vdash_{\mathcal{P}} \left\{{}^{!x}\!/\!{}_{x}\right\} A, \Pi \mid \Delta \rightarrow \sigma}{\Gamma \vdash_{\mathcal{P}} \forall x A, \Pi \mid \Delta \rightarrow \sigma} \qquad \frac{\Gamma \vdash_{\mathcal{P}} [\left\{{}^{?x}\!/\!{}_{x}\right\} A] \Delta \rightarrow \sigma}{\Gamma \vdash_{\mathcal{P}} [\exists x A] \Delta \rightarrow \sigma_{\downarrow}}$$

Eigenvariables $!x, !y, \ldots$ (a.k.a skolem symbols)

In proof-search: meta-variables $?x, ?y, \ldots$

\+ some way to record dependencies between eigen- and meta-variables

- *à la* type theory (e.g. Coq): introduction of metas records existing eigens
- *à la* first-order theorem proving: introduction of eigens records existing metas

(skolem symbols are applied)

This records one graph or its complement. $\Rightarrow$ matter of implementation

Closing branches generates constraints on meta-variables

In pure logic (ProLog, 1st-order tableaux,...): constraints = 1st-order unifiers

In presence of theory... e.g. LRA: constraints can be convex polytops

Need for modular treament of constraints in kernel (in PSYCHE: abstract type)

Branching

Meta-variables are shared between proof-tree branches

Closing branches can no longer be done independently

Symmetric conjunctive branching:

$$\frac{\Gamma \vdash_{\mathcal{P}} A_0, \Pi \mid \Delta \to \sigma_0 \qquad \Gamma \vdash_{\mathcal{P}} A_1, \Pi \mid \Delta \to \sigma_1}{\Gamma \vdash_{\mathcal{P}} A_0 \wedge A_1, \Pi \mid \Delta \to \sigma_0 \wedge \sigma_1}$$

Again, in pure logic, $\sigma_0 \wedge \sigma_1$ = most general unifier of σ_0 and σ_1

in LRA, $\sigma_0 \wedge \sigma_1$ can be polytop intersection

Asymmetric conjunctive branching:

$$\frac{\sigma \to \Gamma \vdash_{\mathcal{P}} A_i, \Pi \mid \Delta \to \sigma' \qquad \sigma' \to \Gamma \vdash_{\mathcal{P}} A_{\neg i}, \Pi \mid \Delta \to \sigma''}{\sigma \to \Gamma \vdash_{\mathcal{P}} A_0 \wedge A_1, \Pi \mid \Delta \to \sigma''}$$

Goal: propagation of constraints through branches generalises Prolog-like propagation of substitutions through branches

Abstraction and modularity

D. Rouhling, A. Mahboubi & SGL axiomatised

- the algebraic structure of constraints (some variant of meet-semilattice)
- the specs of decision procedures called at leaves

for the system to work (and to be equivalent to system w/o meta-variables)

J-M Notin & SGL preparing PSYCHE 2.0 according to the above

Backtrack mechanisms get tricky

We also hope that this architecture can capture triggers-based instantiation mechanisms of SMT

VII. Conclusion

Current plugins and decision procedures are illustrative toys

- DPLL plugin very basic

  (though already improved with known techniques of 2-watched literals, clause learning and restarts)

- LRA decision procedure also basic, 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)
- improve DPLL($\mathcal{T}$) plugins to better handle non-clausal formulae
- proof-terms and classical program extraction

  (focused & polarised systems for classical logic originally designed to identify computational content of classical proofs, in connection to classical realisability).

Thank you!

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

Abstract focusing

PSYCHE 3.0 will also gain a level of abstraction by having logic as a parameter:

Following Zeilberger's work, it is possible to describe the concept of focusing (independently from logical connectives and logical system) as an abstract system $F_\_$ where

- intuitionistic focused sequent calculus
- mono-sided classical focused sequent calculus
- bi-sided classical focused sequent calculus

are instances F_J, F_{K1}, F_{K2}.

Possible to generalise Munch's work on realisability and focusing in classical logic & define generic notion of realisability algebra $RA_\_$ to define realisability models of $F_\_$, RA_J (resp. RA_{K1}, RA_{K2}) forming models of F_J (resp. F_{K1}, F_{K2})

Abstract focusing modulo theories

Lifting in the same abstract way our focused sequent calculus with decision procedures, we get an abstract focusing system modulo theories in the form of a functor $F_{\_}(\vdash [\,], \vdash)$ transforming a pair of derivability relations $(\vdash, \vdash [\,])$ (focused, unfocused) into a more general one $(\vdash' [\,], \vdash')$

(just like DPLL($\mathcal{T}$) transforms a decision procedure $\mathcal{T}$ for conjunction of atoms into a decision procedure for arbitrary CNF)

Thinking in terms of compositionable functors seems a convenient way to combine theories

Conjectures:

- By composing functor $F_{\_}$ with β-normalisation, we should get a functor $F_{\_}^{\beta}$ such that $HOL = (F_{\_}^{\beta})^{\infty}(\text{triv}, \text{triv})$
- By composing functor $F_{\_}$ with rewrite rules, we should get a functor $F_{\_}^{\rightarrow}$ such that $(F_{\_}^{\beta})^{\infty}(\text{triv}, \text{triv})$ does deduction modulo