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

Stéphane Graham-Lengrand

CNRS - École Polytechnique, France

Abstract Psyche is a modular proof-search engine designed for either interactive or automated theorem proving, and aiming at two things: a high level of confidence about the output of the theorem proving process and the ability to apply and combine a wide range of techniques. It addresses the first aim by adopting and extending the LCF architecture to guarantee, using private types, not only the correctness but also the completeness of proof search. It addresses the second by offering a much more appropriate API than just the primitives corresponding to the inference rules of the logic in natural deduction: it uses instead a focused sequent calculus for polarised classical logic. Finally, Psyche features the ability to call decision procedures such as those used in Sat-Modulo-Theories solvers. We therefore illustrate Psyche by using it for SMT-solving.

1 Psyche in brief

Psyche [11], the *Proof-Search factorY for Collaborative HEurisics*, is a modular platform for automated or interactive theorem proving, built on an architecture (similar to LCF) where a small *kernel* interacts with *plugins* and *decision procedures*:

- The kernel is based on a proof-search engine *à la* Prolog, offering an API to perform incremental and goal-directed constructions of proof-trees in (a standard but carefully chosen) Sequent Calculus.
- Psyche can produce proof objects (and print them in LaTeX format).
- Plugins can be programmed to drive the kernel, using its API, through the search space towards an answer *provable* or *not provable*; soundness of the answer only relies on the kernel via the use of a private type for answers (similar to LCF's *theorem* type).
- Plugins can be interactive.
- Psyche offers a *memoisation* feature to help program efficient plugins.
- The kernel is parameterised by a procedure deciding the consistency of collections of literals with respect to a background theory, just as in SAT-modulo-theories (SMT) solvers.

The current version 1.5 of Psyche features a kernel designed for propositional logic modulo theories (same logic as that of $\mathsf{DPLL}(\mathcal{T})$ used in SMT-solving), and decision procedures for the empty theory and Linear Rational Arithmetic (LRA). It is a program of about 4200 lines of OCaml 4.00, using hash-consing in most data structures for efficiency.

2 Motivation

Psyche's architecture is designed for the ambition of allowing various theorem proving techniques (generic or problem-specific) to collaborate on a common platform, whilst giving high confidence in the answers produced.

Interfacing the numerous techniques and tools available for theorem proving is legitimately receiving a lot of attention: Automated Theorem Provers, SAT/SMT-solvers, Proof assistants, etc. While trust is already an issue even for a tool running on its own, it becomes even more of an issue when different tools interact. *Proof-checking* is one way of addressing this, being permissive in the algorithms used for theorem proving, as long as they output some proof objects that can be checked. Another way is the LCF-style [7], where only a small kernel of primitives needs to be trusted, and anything smarter (e.g. the interaction between sophisticated techniques) boils down to calls to the primitives.

In the context of proof-checking, a natural way to interact with different (already implemented) techniques, is the black box approach, where an external tool is called and its output is converted back into a proof that can be checked by the system [2,3]. It is somewhat more surprising that, despite the highly programmable possibilities of the LCF architecture, the most successful integration of automated reasoning techniques in an LCF-based proof assistant such as Isabelle [8] seems to also use variants of the black box approach (as very impressively demonstrated by Sledgehammer) [12,10].

Psyche aims at producing answers that are correct by construction, not having to rely on proof-checking; it therefore adopts the LCF philosophy (although it can produce proof objects), also because having a simple trusted kernel is a convenient starting point for different techniques to collaborate. But the goal here is to open the black boxes and program their algorithms directly with calls to the kernel's API, as plugins for Psyche.

Such a deeper level of integration opens up the perspective of interleaving the use of different techniques: An external tool requires an input problem that it can entirely treat; but implementing the *steps* of its algorithm as small progressions in the search-space covered by the main system, allows more possibilities, such as running the technique *up-to-a-point*, where a switch to another technique may be appropriate (e.g. depending on newly generated goals).

The challenge is for the kernel to offer an appropriate API of proof-search or proof-construction primitives, to allow the efficient implementation of theorem proving techniques as plugins. Most LCF-style systems offer primitives corresponding to the inference rules of Natural deduction, or a Hilbert-style system. This is a very fine-grained level, that leaves most (if not all) of the work to the plugin and makes its implementation cumbersome: these formalisms are more about proof-construction than proof-search.

Psyche makes the choice of a bigger grain, and leaves to the kernel some real proof-search computation, but where no decision needs to be made. For this we use a *focused* sequent calculus $\mathsf{LK}^p(\mathcal{T})$ [4,5], which extends the logic programming paradigm to *polarised* classical logic modulo a background theory $\mathcal{T}$. Polarities and Focusing [6,1] are tools that can be used to describe effective proof-search

strategies in Sequent Calculus, hugely narrowing the search-space offered by Gentzen's original rules. In our case, they also specify a sensible division of labour between PSYCHE's kernel and PSYCHE's plugins, redesigning the standard LCF-style API.

3 Overview and general architecture

The kernel contains the mechanisms for exploring the proof-search space in a sound and complete way, taking into account branching and backtracking. It has no strategy regarding the order in which branches are explored, and this lack of intelligence makes its code rather short. If it reaches a proof, then that proof is correct by construction, and if the entire search space is explored and no proof is found, then the kernel correctly outputs that no proof exists.

The plugins then drive the kernel by specifying in which order the branches of the search space should be explored and to which depth, something that is expected to depend on the kind of problem that is being treated. The quality of the plugin is how fast it drives the kernel towards a answer *Provable*/*NotProvable*.

This already departs from the traditional LCF-style in that some actual proof-search computation is performed in the kernel, not just atomic steps of proof-construction:

In traditional LCF, each inference rule of the logic (as on the right-hand side) will give rise to a primitive of the kernel's API:

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

```
name: thm -> ··· -> thm -> thm
```

In PSYCHE's kernel, such an inference rule will be wrapped in the kernel's unique API primitive:

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

such that `machine(conc)` will trigger the recursive calls `machine(prem_1)`,..., `machine(prem_n)`.

PSYCHE's general architecture is illustrated by its main top-level call (slightly reworded for clarity):

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

PSYCHE has a collection of parsers (currently one for DIMACS and one for SMT-Lib2) and calls the appropriate one on PSYCHE's input. The resulting abstract syntax tree is fed to the kernel's `machine` function that will initiate the search. This produces a value of type `output` that is given to the plugin to work with, so as to finally produce an answer *provable* or *not provable*. This could give the impression that the plugin performs computation after the kernel has finished its work, but this is not quite true, as illustrated by the nature of type `output`:

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

which describes the kernel as a *slot machine*: when it is run, it outputs

- either a final answer *provable* or *not provable*
- or a fake output that represents unfinished computation: in order for computation to continue, the plugin needs to "insert another coin in the slot machine"; proof-search will then resume according to the inserted coin.

To summarise, the kernel performs proof-search as long as there is no decision to be made (on which backtrack may later be needed), and when it hits such a point, it stops and asks for another coin to indicate how to proceed next. The plugin drives the kernel in the exploration of the proof-search space by inserting carefully chosen coins, hoping that one day the machine will stop with the jackpot: a value of the form `Final(...)`.

Now while this architecture somewhat departs from LCF, it does share with it the distrust of anything outside the kernel: when concerned with the soundness of the answer (whichever it be), the plugin is here considered as an adversary, so Psyche defines the type `answer` as abstract, i.e. a private type that only the kernel can inhabit (just like the `thm` type of LCF). Psyche's type

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

can be read by the plugin and the top-level if need be, but cannot be inhabited by them. That way, a plugin cannot cheat about Psyche's answer: the worst it can do is to crash Psyche's runs. In Psyche as in traditional LCF, inhabitation of the abstract type (in case of Psyche, with a value of the form `Provable(...)`) explicitly or implicitly constructs a proof of the statement. But contrary to LCF, Psyche also gives guarantees when the output is *not provable*: it can only occur when the kernel has entirely explored the search-space unsuccessfully.

4 Psyche's Kernel

As described above, the kernel's API has the slot machine as its only primitive, controlled by the *coins* that are inserted in it. In order for efficient plugins to be conveniently programmed, the kernel's primitive needs to accept a rather expressive range of coins that can specify a smart exploration of the search-space. This depends on the inference system that is used in the kernel for the incremental and bottom-up construction of proof-trees, and on identifying the inference rules that the kernel will perform automatically from those that will pause computation and prompt the plugin for new directions.

For this, our sequent calculus $\mathsf{LK}^p(\mathcal{T})$ [4,5] describing classical logic modulo a theory $\mathcal{T}$ uses *polarities* and *focusing* (see e.g. [1]). The connectives and literals of classical logic are tagged with polarities: $+$ and $-$. Polarities do not affect the (classical) provability of formulae, but only the shape of proofs and therefore the structure of the proof-search space. *Focusing* is the phenomenon whereby the inference rules decomposing the connectives of the same polarity can be chained without losing completeness (thus narrowing the search-space), see [5] for a full description. But in brief, focusing organises the proof-search process as an alternance between two kinds of phases: *synchronous* and *asynchronous*.

- An asynchronous phase decomposes the formulae of the sequent whose main connective is negative, using *invertible* inference rules (the premises are provable if the conclusion is): these represent no backtrack point in proof-search.
- A synchronous phase starts with the selection of a positive formula: the formula and its sub-formulae are then decomposed recursively (before doing anything else in the sequent) as long as these remain positive. When these become negative, another asynchronous phase starts.

We use focusing to divide the labour between Psyche's kernel and plugins: The kernel applies the asynchronous steps automatically without any instruction from the plugin, and then stops and asks for another coin describing the next synchronous phase, where smart choices may have to be made (starting with the selection of the positive formula to work on). An important consequence of this division of labour is that **every kernel call terminates**, because the length of each phase is bounded by the size of the formula(e) being decomposed.

The choice of polarities on connectives and literals affects the kernel-plugin interaction. For instance the polarity of $\vee$ will determine whether it is decomposed automatically by the kernel (second rule, asynchronous) or with a smart choice by the plugin (first rule, synchronous): $\dfrac{\Gamma \vdash A_i}{\Gamma \vdash A_1 \vee^+ A_2}$ $\dfrac{\Gamma \vdash A_1, A_2, \Gamma'}{\Gamma \vdash A_1 \vee^- A_2, \Gamma'}$

The code of the kernel is rather small (575 lines) and purely functional. Continuation-Passing-Style (CPS) is used to minimise the use of the stack and provide a natural way to represent the progression of the kernel within the search space: the API function `machine: statement->output` actually wraps the real (tail-)recursive function `search: statement->(output->'a)->'a` with the identity continuation. Continuations are heavily used for branching and backtracking (e.g. when `search` applies a rule with several premises, it makes a recursive call on one of the branches and stacks up the others in the passed continuation; similarly when the plugin chooses to explore one branch, the kernel records the other ones -forcing in the end the entire exploration of the search-space), and naturally implement a slot machine waiting for its coin.

5 Plugins

A plugin is an OCaml module of a fixed module type declaring a function `solve: output->answer` (again, `answer` is for the plugin an abstract type).

However, it is likey that the sophisticated strategies/heuristics that the plugin is meant to implement rely on some clever choice of data-structures for formulae, sets of formulae, sets of literals. So the plugin and the kernel have to agree on those three data-structures that are communicated both ways during the interaction. In Psyche 1.5, the kernel is parameterised by the data-structures, and the plugin provides them.

We first tested Psyche's architecture with a basic plugin `Naive`, which implements collections as lists and inserts the first available coin in the slot machine, whenever asked. This works fine for small tautologies, printable on a screen.

But the first real aim was to capture in Psyche some propositional SAT and SAT-Modulo-Theories solving techniques, making $\mathsf{DPLL}(\mathcal{T})$ technology available in a generic proof-search framework like Psyche. For this we describe in [5] how to see $\mathsf{DPLL}(\mathcal{T})$, canonically expressed as a transition system [9], as a simple bottom-up proof-construction mechanism in $\mathsf{LK}^p(\mathcal{T})$. More practically, every rule of $\mathsf{DPLL}(\mathcal{T})$ can be seen as the insertion of a particular coin in Psyche's slot machine. We implemented this as two different plugins for Psyche: `DPLL_Pat`

and `DPLL_WL`. These remain toy plugins, because, although it is now clear from [5] how to perform each rule of DPLL($\mathcal{T}$) in Psyche, we still have to decide which rule to apply. This is where the two plugins differ: `DPLL_Pat` looks up the applicability of DPLL($\mathcal{T}$)'s rules by using Patricia tries to implement sets of clauses, while `DPLL_WL` looks it up using the technique of *watched literals*.

Just like DPLL($\mathcal{T}$)-based solvers are made efficient by using features such as backjumping and lemma learning, our plugins are made more efficient by the use of memoisation, which avoids re-doing, for some open branch, the same steps as those used in a previously completed branch. Psyche 1.5 therefore offers a memoisation module, to be used by plugins to record values of (the abstract) type `answer`. And the kernel's slot machine accepts from the plugin, as a special coin carrying such a value, "here is an already found answer that also applies to the current goal". The kernel accepts the value as closing the current branch (one way or another) **without any proof-checking** (since the abstract type ensures the value came as an earlier output of the kernel); it only checks that the value applies to the current goal. Now for a memoised answer *Provable* to be reusable as often as possible, it is useful to prune the provable sequent, just before it is tabled, from the formulae and literals that were not used in its proof. This is easy to do for the complete proofs of $\mathsf{LK}^p(\mathcal{T})$ (eager weakening are applied a posteriori by inspection of the inductive structure). Psyche's kernel actually performs the pruning on-the-fly whenever an inference is added to complete proofs, so that, whenever it outputs `Final(sequent,proof)`, the sequent is already pruned. This is Psyche's way of performing *conflict analysis*, a key process of SMT-solving.

6 Conclusion and perspectives

Psyche is run from the command-line, taking as input the indicated file(s) or directory(ies) (or the standard input): `psyche [OPTION]... [FILE/DIR]...` Version 1.5 is distributed with a DIMACS parser, which we used to test Psyche on (propositional) SAT benchmarks, and an SMTLib2 parser (unmodified from the Alt-Ergo prover), which we used to test it on `QF_LRA` benchmarks (making use of the distributed simplex algorithm for LRA). The results are available on Psyche's website [11]. Since Psyche has no ambition to beat state-of-the-art SAT- and SMT-solvers, it works well on small instances but its performance starts declining between 20Kb and 100Kb of input problem size (of course this is no appropriate measure of difficulty): There is no intrinsic problem of scalability, but the current plugins and decision procedures are illustrative toys (the heuristics for applying DPLL($\mathcal{T}$) rules in the current plugins are still basic, and so is the decision procedure for LRA -e.g. it is not incremental). What we offer here is a platform and its modularity: anyone with better (or different) heuristics or decision procedures can simply write them as OCaml modules of our predefined module types, and Psyche will seamlessly run with them, keeping the same LCF-style guarantees. Moreover, nothing in Psyche's proof-engine relies on the input being sets of clauses, so Psyche might offer a convenient framework to generalise the known techniques for the satisfiability of formulae in clausal form.

The short-to-medium term plans are as follows:

- **kernel**: handle existential variables and propagate first-order unifiers through branching, construct and store proof-terms rather than whole proofs;
- **theories**: improve the procedure for LRA (e.g. making it incremental) and implement other theories (Congruence Closure, LIA, bit vectors, etc);
- **plugins**: implement a user-interactive plugin asking which coins to insert, improve the DPLL($\mathcal{T}$) plugins to better handle non-clausal formulae, and implement other theorem proving techniques as plugins: *analytic tableaux*, *clausal tableaux* (e.g. *connections*) and *resolution* are all done in the theory.

In the long-term, we plan to investigate whether $\mathsf{LK}^p(\mathcal{T})$ may help mixing first-order reasoning with theories (i.e. investigate instantiations in presence of a theory), and prove Psyche's correctness in a proof assistant (since the functional kernel seems small enough and the plugins need not be certified).

Acknowledgements. Contributors of Psyche include S. Graham-Lengrand, A. Mahboubi, A. Bernadet, M. Farooque, Damien Rouhling and M. Vegreville.

References

1. J. M. Andreoli. Logic programming with focusing proofs in linear logic. *J. Logic Comput.*, 2(3):297–347, 1992.
2. Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. A modular integration of SAT/SMT solvers to Coq through proof witnesses. In *Proc. of the 1st Int. Conf. on Certified Programs and Proofs (CPP'11)*, volume 7086 of *LNCS*, pages 135–150. Springer, December 2011.
3. Frédéric Besson, Pierre-Emmanuel Cornilleau, and David Pichardie. Modular SMT proofs for fast reflexive checking inside Coq. In Jean-Pierre Jouannaud and Zhong Shao, editors, *Certified Programs and Proofs*, volume 7086 of *LNCS*, pages 151–166. Springer-Verlag, 2011.
4. Mahfuza Farooque and Stéphane Graham-Lengrand. Sequent calculi with procedure calls. Technical report, Laboratoire d'Informatique de l'Ecole Polytechnique, January 2013. http://hal.archives-ouvertes.fr/hal-00779199
5. Mahfuza Farooque, Stéphane Graham-Lengrand, and Assia Mahboubi. A bisimulation between DPLL(T) and a proof-search strategy for the focused sequent calculus, June 2013. Available on Psyche's website [11].
6. Jean-Yves Girard. Linear logic. *Theoret. Comput. Sci.*, 50(1):1–101, 1987.
7. Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: a mechanized logic of computation*, volume 78 of *LNCS*. Springer-Verlag, 1979.
8. The Isabelle theorem prover. http://isabelle.in.tum.de/
9. Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. of the ACM Press*, 53(6):937–977, 2006.
10. Lawrence C. Paulson and Jasmin Christian Blanchette. Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In Geoff Sutcliffe, Stephan Schulz, and Eugenia Ternovska, editors, *IWIL 2010*, volume 2 of *EPiC Series*, pages 1–11. EasyChair, 2012.
11. Psyche: the Proof-Search factorY for Collaborative HEuristics. http://www.lix.polytechnique.fr/~lengrand/Psyche/
12. Tjark Weber. SMT solvers: New oracles for the HOL theorem prover. *International Journal on Software Tools for Technology Transfer (STTT)*, 13(5):419–429, 2011.