The Pocket Reasoner – Automatic Reasoning on Small Devices

Jens Otten

University of Oslo

`jeotten@ifi.uio.no`

Abstract

Automated reasoning in classical first-order logic is a core research field in Artificial Intelligence. Most of the fully automated reasoning tools are large and complex systems implementing proof search methods that have significant memory requirements. This paper presents an automated reasoning tool implemented on an iPod Nano. It is based on leanCoP, a very compact Prolog implementation of the connection calculus, which operates on the structure of the given formula without generating new subformula instances. Hence, the memory requirements are significantly lower, allowing leanCoP to run on devices with only little (random-access) memory. The paper presents details of the proof search calculus, its implementation, and a practical evaluation of the presented reasoning tool.

1 Introduction

Logical reasoning is a fundamental task, not only in mathematics and computer science, but in everyday life. *Automated Theorem Proving* (ATP) is a core research field in Artificial Intelligence that aims to automate formal logical reasoning. Over the last decades the development of reasoning tools, so called *(theorem) provers*, has made significant progress; see [17] for an overview. Originally designed to assist mathematicians when proving theorems, nowadays, automatic and interactive theorem provers have many important applications, e.g., in hardware and software verification [7, 16, 18] and program synthesis [4].

ATP is concerned with the question whether a conjecture G is a *logical consequence* of a given set of axioms $\{F_1,\ldots,F_n\}$, written $F_1,\ldots,F_n \models G$, in which $F_1,\ldots,F_n$ and G are first-order formulae. According to the *deduction theorem* (for classical logic), $F_1,\ldots,F_n \models G$ holds if (and only if) the formula $(F_1 \wedge \ldots \wedge F_n) \Rightarrow G$ is valid. Hence, the objective of an ATP system is to determine if a given formula F is *valid* (with respect to a specific logic); in this case the ATP system usually also outputs a *proof* of F. A formula F is *valid* if and only if it evaluates to *true* for all possible interpretation of its function and predicate symbols.

$$\begin{tikzcd}
 & & \text{F is \textit{valid} (proof)} \\
\begin{array}{c}\text{formula } F\\ \text{(problem)}\end{array} \arrow[r] & \boxed{\begin{array}{c}\text{ATP system}\\ \text{(``prover'')}\end{array}} \arrow[ur] \arrow[dr] & \\
 & & \text{F is \textit{not} valid (counter model)}
\end{tikzcd}$$

This paper was presented at the NIK-2018 conference; see http://www.nik.no/.

For example, if we know that *Plato is a man*, and *all men are mortal*, then, as a logical consequence, *Plato is mortal*. In the language of first-order logic this is represented by the following formula

$$(man(Plato) \wedge \forall X\,(man(X) \Rightarrow mortal(X))) \Rightarrow mortal(Plato)\,.$$

A *formal* description of the proof search algorithm is usually specified in form of a *(proof) calculus* consisting of axioms and rules. Many different proof calculi exist, most notable the resolution calculus [15], which was specifically developed with the automation on computers in mind. The main rule of this calculus takes two clauses of the input formula and derives a new clause, which is added to the input clauses. Another popular approach are instance-based methods [8], which iteratively generate ground instances of first-order clauses and then use a solver for propositional logic. While both of these approaches have successfully been implemented, their main disadvantage is that they generate thousands or even millions of new clauses during the proof search. Hence, ATP systems based on these calculi usually need large amounts of memory.

This paper presents an approach for automating logical reasoning that does not have this disadvantage. It is based on the connection calculus and implemented within a few lines of Prolog code. As a proof-of-concept and to show the low memory requirements, the ATP system is adapted to run on an iPod Nano, a compact, portable media player.

The paper is organized as follows. Section 2 introduces some basic concepts of first-order logic and the connection calculus for classical first-order logic. Section 3 presents a very compact Prolog implementation of this calculus. Section 4 describes the installation of the prover on an iPod Nano and Section 5 presents some examples that show the reasoning capabilities of the presented reasoning tool and an analysis of the memory usage. Section 6 concludes with a summary and a brief outlook on further research.

2 Connection-based Reasoning

The implementation of the reasoning tool, i.e., of the ATP system presented later on is based on the *connection calculus*. Such a (proof) calculus provides a formal description of a method to determine the validity of a given first-order formula. The (clausal) connection calculus requires the input formula to be in a *clausal* or *matrix* representation.

First-Order Logic

The standard language of (classical) *first-order logic* is used for representing the given input formulae, see, e.g., [1, 20]. The letter X is used to denote variables, and terms are built from functions, constants and variables.

An *atomic formula*, denoted by A, is built from predicate symbols and terms. The connectives $\neg$, $\wedge$, $\vee$ and $\Rightarrow$ denote negation, conjunction, disjunction and implication, respectively. A *(first-order) formula*, denoted by F consists of atomic formulae, the connectives and the existential and universal quantifiers $\exists$ and $\forall$. For example,

$$(man(Plato) \wedge \forall X\,(man(X) \Rightarrow mortal(X))) \Rightarrow mortal(Plato) \qquad (1)$$

is a first-order formula.

A *literal* is an atomic formula or a negated atomic formula, i.e. it has the form A or $\neg A$. The *complement* $\overline{L}$ of a literal L is A if L is of the form $\neg A$, and $\neg L$ otherwise.

Matrix Representation

The *matrix representation* is used for first-order formulae in clausal form. A *clause* C has the form $(L_1 \wedge L_2 \wedge \ldots \wedge L_m)$, where each L_i is a literal. A formula in *(disjunctive) clausal form* has the form $\exists x_1 \ldots \exists x_n \, (C_1 \vee \ldots \vee C_n)$, where each C_i is a clause.[1] For example, formula (1) has the equivalent clausal form

$$\exists X \, (\neg man\,(Plato) \vee (\, man\,(X) \wedge \neg mortal\,(X)\,) \vee mortal\,(Plato)\,)\;. \qquad (2)$$

For classical (first-order) logic, every formula F can be translated into an equivalent formula F' in clausal form, such that F is valid *iff* (if and only if) F' is valid. A set of clauses is represented as a matrix. A *matrix* of a formula consists of its clauses $\{C_1, \ldots, C_n\}$, in which each *clause* is a set of its literals $\{L_1, \ldots, L_m\}$. A *polarity* 0 or 1 is used to represent negation in a matrix, i.e. literals of the form A and $\neg A$ are represented by A^0 and A^1, respectively, In the *graphical representation* of a matrix, its clauses are arranged horizontally, while the literals of each clause are arranged vertically. For example, the matrix of formula (2) is

$$M = \{\{man\,(Plato)^1\}, \{man\,(X)^0, mortal\,(X)^1\}, \{mortal\,(Plato)^0\}\} \qquad (3)$$

which has the graphical representation

$$\left[\ \left[\ man\,(Plato)^1\ \right] \ \left[\begin{array}{c} man\,(X)^0 \\ mortal\,(X)^1 \end{array}\right] \ \left[\ mortal\,(Plato)^0\ \right]\ \right]\;. \qquad (4)$$

A *connection* is a set $\{A^0, A^1\}$ of literals with the same predicate symbol but different polarities. A *term substitution* σ assigns terms to variables that occur in the literals of a given formula. A connection $\{L_1, L_2\}$ with $\sigma(L_1) = \sigma(\overline{L_2})$ is called *σ-complementary*. For example, the connection $\{man\,(X)^0, man\,(Plato)^1\}$ is a σ-complementary connection, if the variable X is substituted by *Plato*, i.e. for $\sigma(X) = Plato$.

A *path* through a matrix $M = \{C_1, \ldots, C_n\}$ is a set of literals that contains one literal from each clause $C_i \in M$, i.e. a set $\cup_{i=1}^{n} \{L_i'\}$ with $L_i' \in C_i$. For example, $\{man\,(Plato)^1$, $man\,(X)^0$, $mortal\,(Plato)^0\}$ and $\{man\,(Plato)^1$, $mortal\,(X)^1$, $mortal\,(Plato)^0\}$ are (the only) paths through matrix (3) or its graphical representation (4).

According to the the *matrix characterization*, a formula F is valid iff there exists a term substitution σ and a set of connections S, such that every path through the matrix M of F contains a σ-complementary connection $\{L_1, L_2\} \in S$; see [3] for details.

Connection Calculus

The connection calculus [2, 3], is a well-known basis to automate formal reasoning in classical first-order logic. Proof search in the connection calculus is guided by connections in order to calculate an appropriate set of connections S (according to the matrix characterization).

In each step of a derivation in the connection calculus a connection is identified and only paths that do not contain this connection are investigated afterwards. If every path contains a σ-complementary connection, the proof search succeeds and the given formula

[1]Even though the use of a (negated) *conjunctive clausal form* is very common, a disjunctive clausal form is used for historical and practical reasons; the difference between both forms is marginal.

$$\begin{array}{rll}
\textit{Axiom (A)} & \dfrac{}{\{\},M,Path} & \\[2ex]
\textit{Start (S)} & \dfrac{C_2,M,\{\}}{\varepsilon,M,\varepsilon} & \text{and } C_2 \text{ is copy of } C_1{\in}M \\[2ex]
\textit{Reduction (R)} & \dfrac{C,M,Path{\cup}\{L_2\}}{C{\cup}\{L_1\},M,Path{\cup}\{L_2\}} & \text{and } \sigma(L_1)=\sigma(\overline{L_2}) \\[2ex]
\textit{Extension (E)} & \dfrac{C_2{\setminus}\{L_2\},M,Path{\cup}\{L_1\} \quad C,M,Path}{C{\cup}\{L_1\},M,Path} & \text{and } C_2 \text{ is a copy of } C_1{\in}M,\ L_2{\in}C_2, \text{ and } \sigma(L_1)=\sigma(\overline{L_2})
\end{array}$$

Figure 1: The clausal connection calculus

is valid. A *connection proof* can be illustrated within the graphical matrix representation. For example, the proof of matrix 3 consists of two steps, which identify two connections:

$$\left[\ \left[\ man\,(Plato)^1\ \right]\ \left[\begin{array}{c} man\,(X)^0 \\ mortal\,(X)^1 \end{array}\right]\ \left[\ mortal\,(Plato)^0\ \right]\ \right]. \tag{5}$$

In contrast to sequent or (standard) tableau calculi, the *connection-driven* search strategy permits a more *goal-oriented* proof search. This leads to a significantly smaller search space and, thus, to a more efficient proof search. A connection corresponds to a closed branch in the tableau calculus [6] or an axiom in the sequent calculus [5].

The formal description of the connection calculus is given in Fig. 1 and consists of one axiom and three rules [13]. It works on tuples of the form "$C,M,Path$", where M is a matrix, C is the *subgoal clause* and *Path* is the *active path*; σ is a *rigid* term substitution. A *connection proof* of a matrix M is a clausal connection proof of $\varepsilon,M,\varepsilon$. For example,

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{\},M,\{mortal(Plato)^0,man(X')^0\}}A \qquad \dfrac{}{\{\},M,\{mortal(Plato)^0\}}A}{\{\mathbf{man(X')^0}\},\{\{\mathbf{man(Plato)^1}\},\ldots\},\{mortal(Plato)^0\}}E \qquad \dfrac{}{\{\},M,\{\}}A}{\{\mathbf{mortal(Plato)^0}\},\{\{man(Plato)^1\},\{man(X)^0,\mathbf{mortal(X)^1}\},\{mortal(Plato)^0\}\},\{\}}E}{\varepsilon,\{\{man(Plato)^1\},\{man(X)^0,mortal(X)^1\},\{mortal(Plato)^0\}\},\varepsilon}S$$

is a (formal) connection proof of matrix (3) with the term substitution $\sigma(X')=Plato$ (the variable X' occurs in the copy of the second clause).

Proof *search* in the clausal connection calculus is carried out by applying the rules of the calculus in an *analytic* way, i.e. from bottom to top, starting with $\varepsilon,M,\varepsilon$, in which M is the matrix of the given formula. At first, a start clause is selected. Afterwards, connections are successively identified by applying reduction and extension rules, in order to make sure that all paths through the matrix M contain a σ-complementary connection. This process is guided by the active path, a subset of a path through M. During the proof search, *backtracking* might be required when choosing the clause C_1 in the start and extension rules or the literal L_2 in the reduction and extension rules, in case the chosen rule or rule instance does not lead to a proof. The term substitution σ is calculated step by step by one of the well-known *term unification* algorithms (see, e.g. [15]) whenever a reduction or extension rule is applied.

3 The Prolog Implementation

leanCoP is a very compact Prolog implementation of the connection calculus described in Sect. 2. leanCoP 1.0 essentially implements the basic connection calculus shown in Fig. 1 [13]. leanCoP 2.0 integrates additional optimization techniques into the basic connection calculus [10]. leanCoP 2.1 adds a proof output and a fixed strategy scheduling. The source code of the leanCoP 2.1 core prover is shown in Fig. 2.

```
prove(PathLim,Set,Proof) :-
    \+member(scut,Set) -> prove([-(#)],[],PathLim,[],Set,[Proof]) ;
    lit(#,C,_) -> prove(C,[-(#)],PathLim,[],Set,Proof1),
    Proof=[C|Proof1].
prove(PathLim,Set,Proof) :-
    member(comp(Limit),Set), PathLim=Limit -> prove(1,[],Proof) ;
    (member(comp(_),Set);retract(pathlim)) ->
    PathLim1 is PathLim+1, prove(PathLim1,Set,Proof).

prove([],_,_,_,_,[]).
prove([Lit|Cla],Path,PathLim,Lem,Set,Proof) :-
    Proof=[[[NegLit|Cla1]|Proof1]|Proof2],
    \+ (member(LitC,[Lit|Cla]), member(LitP,Path), LitC==LitP),
    (-NegLit=Lit;-Lit=NegLit) ->
       ( member(LitL,Lem), Lit==LitL, Cla1=[], Proof1=[]
         ;
         member(NegL,Path), unify_with_occurs_check(NegL,NegLit),
         Cla1=[], Proof1=[]
         ;
         lit(NegLit,Cla1,Grnd1),
         ( Grnd1=g -> true ; length(Path,K), K<PathLim -> true ;
           \+ pathlim -> assert(pathlim), fail ),
         prove(Cla1,[Lit|Path],PathLim,Lem,Set,Proof1)
       ),
       ( member(cut,Set) -> ! ; true ),
       prove(Cla,Path,PathLim,[Lit|Lem],Set,Proof2).
```

Figure 2: The complete source code of the leanCoP 2.1 core prover

Prolog lists are used to represent sets, Prolog terms and variables are used to represent atomic formulae and term variables, respectively; "-" is used to mark literals that have polarity 1. The logical connectives and quantifiers are expressed by "~" (negation), "," (conjunction), ";" (disjunction), "=>" (implication), `ex X:` (existential quantifier), and `all X:` (universal quantifier). For example, formula (1) is represented by the Prolog term

```
( man(plato) , all X: ( man(X) => mortal(X) ) ) => mortal(plato) .
```

In a preprocessing step, the input formula is translated into a matrix M and stored in Prolog's database in the following form. For every clause $C \in M$ and for all literals $\mathtt{L} \in C$ the fact "`lit(L,C1,Grnd)`" is stored, where $\mathtt{C1} = C \setminus \{\mathtt{L}\}$ and `Grnd` is `g` if C is ground (i.e. does not contain any term variables) and `n`, otherwise. For example, the three clauses (cl. 1–3) of matrix (3) are stored in the following form:

```
lit(-man(Plato),[],g).     (cl. 1)     lit(man(X),[-mortal(X)],n).  (cl. 2)
lit(mortal(Plato),[],g).   (cl. 3)     lit(-mortal(X),[man(X)],n).  (cl. 2)
```

This integrates the main advantage of the ”Prolog technology” approach [21] into leanCoP by using Prolog’s fast indexing mechanism to quickly find connections.

The core prover in Fig. 2 is invoked with `prove(1,Set,Proof)`, where `Set` is a strategy and the initial limit for the size of the active path is 1; see also [10]. The predicate succeeds iff there is a connection proof for the matrix/clauses stored in Prolog’s database. The proof search starts by applying the start rule (lines a–d). The path limit is used to perform iterative deepening on the size of the active path (lines e–h), which is necessary for completeness. Afterwards, reduction rule (lines 2–3,5,8–9,17) and extension rule (lines 2–3,5,11–14,17) are repeatedly applied, until a branch in the derivation can be closed by an axiom (line 1). The term substitution σ is stored implicitly by Prolog.

Several proof search optimizations are integrated into the prover. *positive start clauses*, *regularity* (line 4), *lemmata* (line 6), and *restricted backtracking* (line 16); see [9, 10] for details. Furthermore, leanCoP 2.1 uses a *fixed strategy scheduling*, i.e. a shell script invokes the Prolog core prover, consecutively, with different strategies.

The full source code of the prover, including the clausal form translation, are available on the leanCoP website at `http://www.leancop.de`.

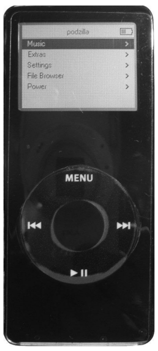
Other versions of leanCoP include implementations for first-order intuitionistic and first-order modal logic. They are based on an adapted matrix characterization for non-classical logics [23]; see [14] for an overview of these provers.

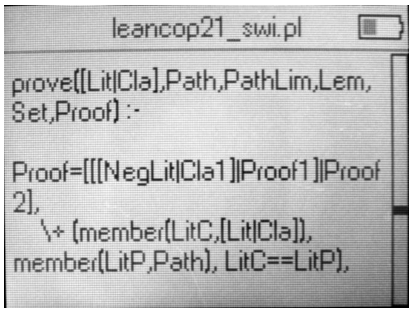
4 A Pocket Reasoning Tool

In order to show the modest hardware requirements of the leanCoP 2.1 prover presented in Sect. 3, it has been ported and installed on an iPod Nano. See Table 1 for an overview.

Hardware

The (first-generation) iPod Nano is a very compact portable media/MP3 player that was released in 2005. It measures only 40 mm x 90 mm x 6.9 mm (24.8 cm^3) and weights only 42 grams (see also Fig. 3). The iPod Nano uses a PortalPlayer PP5021C "system on a chip" with dual embedded 80 MHz ARM 7TDMI processors (CPUs). The ARM 7TDMI is a 32-bit RISC CPU and one of the most widely used ARM cores, which can be found in numerous embedded systems. It has 32 MB of random access memory (RAM), but *no* Memory Management Unit (MMU), which is necessary to use and address virtual memory. The storage capacity is between 1 GB and 4 GB of flash memory (the iPod version used in the following has 2 GB). The iPod Nano has a small 1.5 inch (16-bit) colour screen with a 176 x 132 resolution. Navigation is done by a small “click wheel”.




leancop21_swi.pl

```
prove([Lit|Cla],Path,PathLim,Lem,
Set,Proof) :-

Proof=[[[NegLit|Cla1]|Proof1]|Proof
2],
  \+ (member(LitC,[Lit|Cla]),
member(LitP,Path), LitC==LitP),
```

Figure 3: iPod Nano running μClinux and screenshot showing (part of) the leanCoP code

Software

In order to run the leanCoP prover on the iPod Nano, a Prolog implementation is installed, which runs on the iPodLinux operating system. μClinux (or "uClinux") is a derivative of Linux intended for microcontrollers without Memory Management Unit (MMU). iPodLinux is a μClinux-based Linux distribution designed specifically to run on the iPod, including the first-generation iPod Nano.[2] When the iPodLinux kernel is booted, it runs instead of the original iPod operating system and invokes Podzilla, an alternative graphical user interface, allowing the user to interact with the device and launch other applications.

SWI-Prolog version 5.0.10 Lite is an efficient, portable ISO-standard Prolog.[3] The SWI-Prolog runtime environment `plrt` was built using the gcc 3.4.3 cross compiler of the ARM-μClinux tool chain. The SWI-Prolog start-up file `plrt.prc` is platform independent and was built on an x86-Linux platform.

As the μClinux shell does not allow the execution of shell scripts, the prover has to be invoked by an executable *problem binary* `<problem-bin>`. This file is a compiled C source file that only contains a system call to load and run the Prolog runtime environment and the leanCoP prover. The strategy scheduling realized within leanCoP's shell script can not be used anymore. The following files are required to run leanCoP on the iPod Nano:

- `leancop21_swi.pl` (3.3 KB): the leanCoP core prover (see Fig. 2)
- `def_mm.pl` (7.2 KB): module for the translation into clausal form
- `leancop_main.pl` (2.9 KB): main module that invokes the different Prolog modules
- `leancop_proof.pl` (8.0 KB): translates returned proof into a readable form
- `leancop_tptp2.pl` (5.1 KB): translates the TPTP input syntax into leanCoP syntax
- `<problem-bin>` (4.8 KB): executable that calls the main Prolog module
- `plrt` (657.7 KB): the SWI-Prolog runtime environment
- `plrt.prc` (77.0 KB): the SWI-Prolog start-up file (containing Prolog libraries)

Overall, the files have a size of 766 KB. From the total available RAM of 24 MB (after booting and starting podzilla), SWI-Prolog and leanCoP need only about 1 MB when loaded and several MBs when the prover is invoked (see also Sect. 5). Whereas provers based on resolution or instance-based calculi need to store many of inferred new clauses in memory, leanCoP needs to store only the following data: (a) the given matrix/clauses (see Sect. 3), (b) the subgoal clause *C* and the active path *Path* (see Sect. 2), and (c) the stack required by SWI-Prolog for the recursive calls of the `prove` predicate.

Hardware:	**Software:**
• iPod Nano (first generation)	• μClinux: small Linux for embedded systems (without MMU)
• Dimensions: 40 x 90 x 6.9mm; 42 grams	• SWI-Prolog 5.0.10 Lite: efficient, portable ISO-standard Prolog; built by using gcc ARM cross-compiler
• PortalPlayer PP5021C ("system on a chip")	• leanCoP 2.1 prover (with proof output, but without strategy scheduling)
• CPU: dual embedded 80 MHz ARM 7TDMI, 32-bit RISC	
• RAM: 32 MB, *no* memory management Unit (MMU)	

Table 1: Technical details of the pocket reasoning tool

[2]http://www.ipodlinux.org/; due to encrypted firmware, later iPod Nano generations are not supported.

[3]http://www.swi-prolog.org/; about a dozen different Prolog implementations were considered for running on the iPod Nano; only SWI-Prolog showed a stable and good performance.

5 Practical Evaluation

The following two examples, a logical puzzle and a theorem of mathematical set theory, show how the pocket reasoner presented in Sect. 4 can be used to automate formal reasoning. An analysis of the memory usage is presented as well.

Example 1: Who Killed Aunt Agatha?

Consider the following puzzle:

> *(1) Someone who lives in Dreadbury Mansion killed Aunt Agatha. (2–4) Agatha, the butler, and Charles live in Dreadbury Mansion, (5) and are the only people who live therein. (6) A killer always hates his victim, (7) and is never richer than his victim. (8) Charles hates no one that Aunt Agatha hates. (9) Agatha hates everyone except the butler. (10) The butler hates everyone not richer than Aunt Agatha. (11) The butler hates everyone Aunt Agatha hates. (12) No one hates everyone. (13) Agatha is not the butler. (14) Therefore: Did Agatha kill herself?*

A formalization of this puzzle is given in problem file PUZ001+1 of the TPTP library [22] and shown in Fig. 4. The (shortened) header contains file, domain and problem names, the (known) status of the problem and its difficulty rating, expressed by a number between 0 (easy) and 1 (very difficult). E.g., the shown rating of 0.07 indicates that 0.07% of all state-of-the-art provers are *not* able to solve this problem.

Lines 1–13 contain the axioms of the problem, line 14 contains the conjecture (the numbering is added for convenience only). Each axiom or conjecture is represented by a first-order formula in TPTP syntax, using the logical connectives `~` (negation), `&` (conjunction), `|` (disjunction), `=>` (implication), and the quantifiers `?[X]:` ($\exists X$) and `![X]:` ($\forall X$). The problem also contains equality (`=`) and inequality (`!=`); in this case leanCoP automatically adds the necessary equality axioms.

```
     %------------------------------------------------------------------
     % File    : PUZ001+1 : TPTP v6.4.0. Released v2.0.0.
     % Domain  : Puzzles
     % Problem : Dreadbury Mansion
     % Status  : Theorem
     % Rating  : 0.07 v6.4.0, 0.12 v6.3.0, 0.04 v6.2.0, 0.12 v6.1.0, 0.20 v
     %------------------------------------------------------------------

( 1) fof(pel55_1,axiom,   ( ? [X]:( lives(X) & killed(X,agatha) ) )).
( 2) fof(pel55_2_1,axiom, ( lives(agatha) )).
( 3) fof(pel55_2_2,axiom, ( lives(butler) )).
( 4) fof(pel55_2_3,axiom, ( lives(charles) )).
( 5) fof(pel55_3,axiom, ( ! [X]:(lives(X)=>(X=agatha|X=butler|X=charles)))).
( 6) fof(pel55_4,axiom, ( ! [X,Y]:( killed(X,Y) => hates(X,Y) ) )).
( 7) fof(pel55_5,axiom, ( ! [X,Y]:( killed(X,Y) => ~richer(X,Y) ) )).
( 8) fof(pel55_6,axiom, ( ! [X]:( hates(agatha,X) => ~hates(charles,X) ))).
( 9) fof(pel55_7,axiom, ( ! [X]:( X != butler => hates(agatha,X) ) )).
(10) fof(pel55_8,axiom, ( ! [X]:( ~richer(X,agatha) => hates(butler,X) ))).
(11) fof(pel55_9,axiom, ( ! [X]:( hates(agatha,X) => hates(butler,X) ) )).
(12) fof(pel55_10,axiom, ( ! [X]: ? [Y]: ~hates(X,Y) )).
(13) fof(pel55_11,axiom, ( agatha != butler )).
(14) fof(pel55,conjecture, ( killed(agatha,agatha) )).
```

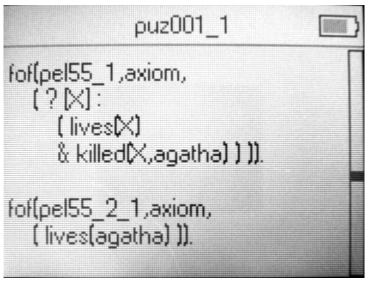
Figure 4: Formalization of “Who killed Aunt Agatha?”

```
1 Assume killed(agatha, agatha) is false.
  Then clause (22) under the substitution [[_G654,_G655,_G656,_G657], [1^[],
  agatha,agatha,agatha]]
  is false if at least one of the following is false:
  [killed(1^[], agatha), 1^[]=agatha, agatha=agatha]
1.1 Assume killed(1^[], agatha) is false.
    Then clause (3) is true.
1.2 Assume 1^[]=agatha is false.
    Then clause (7) under the substitution [[_G676], [1^[]]]
    is false if at least one of the following is false:
    [lives(1^[]), -1^[]=butler, -1^[]=charles]
1.2.1 Assume lives(1^[]) is false.
      Then clause (2) is true.
1.2.2 Assume -1^[]=butler is false.
      Then clause (17) under the substitution [[_G594,_G593], [butler,1^[]]]
      is false if at least one of the following is false:
      [-butler=1^[]]
[...]
1.3.3.1.2 Assume agatha=charles is false.
          This is a contradiction to assumption 1.3.3.

Therefore there is no interpretation that makes all
given clauses simultaneously false. Hence the given
clauses are valid.
                                        q.e.d.
```

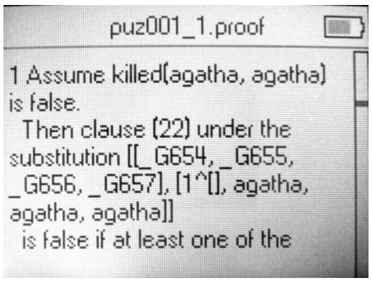
Figure 5: Solution/proof for the "Who killed Aunt Agatha?" puzzle

It takes leanCoP on the iPod Nano about 14 seconds to output a solution/proof for this problem. The proof consists of 46 proof steps (i.e. applications of start, reduction and extension rules). The first five and the last steps are shown in Fig. 5; they closely follow the structure of a connection proof. Terms starting with an underscore are variables; terms of the form `I^[]` are skolemized variables introduced during the translation into clausal form. These clauses are part of the proof output, but not shown Fig. 5. Screenshots of (part of) the problem representation and the generated proof are shown in Fig. 6.



```
puz001_1
fof(pel55_1,axiom,
   ( ? [X] :
      ( lives(X)
      & killed(X,agatha) ) )).

fof(pel55_2_1,axiom,
   ( lives(agatha) )).
```



```
puz001_1.proof
1 Assume killed(agatha, agatha)
is false.
 Then clause (22) under the
substitution [[_G654, _G655,
_G656, _G657], [1^[], agatha,
agatha, agatha]]
 is false if at least one of the
```

Figure 6: Problem file PUZ001+1 and its proof on the iPod Nano

Example 2: Intersection Distributes Over Union

Problem SET169+3 from the TPTP problem library states that, given three sets, intersection distributes over union, i.e.

$$\forall B\, \forall C\, \forall D: \quad B \cap (C \cup D) = (B \cap C) \cup (B \cap D)\ . \tag{6}$$

The formalization of this theorem in TPTP syntax is presented in Fig. 7. Again, the file starts with a header showing file, domain, and problem names, as well as status and rating. The (current) rating of 0.60 indicates that this problem is *not* solved by 60% of the state-of-the-art theorem provers.

```
     %------------------------------------------------------------------------
     % File     : SET169+3 : TPTP v6.4.0. Released v2.2.0.
     % Domain   : Set Theory
     % Problem  : Intersection distributes over union
     % Status   : Theorem
     % Rating   : 0.60 v6.4.0, 0.62 v6.3.0, 0.67 v6.2.0, 0.68 v6.1.0, 0.73
     %------------------------------------------------------------------------

( 1) fof(union_defn,axiom,
( 2) [...]
( 3) fof(intersection_defn,axiom,
( 4)    ( ! [B,C,D]:
( 5)       ( member(D,intersection(B,C))<=>(member(D,B) & member(D,C)) ) )).
( 6) [...]
( 7) fof(subset_defn,axiom,
( 8)     ( ! [B,C]: ( subset(B,C) <=>
( 9)                  ! [D] : ( member(D,B)  => member(D,C) ) ) )).
(10) [...]
(11) fof(prove_intersection_distributes_over_union,conjecture,
(12)     ( ! [B,C,D]: intersection(B,union(C,D)) =
(13)                  union(intersection(B,C),intersection(B,D)) )).
```

Figure 7: Formalization of "Intersection distributes over union"

Three of the axioms and the conjecture are shown in Fig. 7. For example, the second axiom (lines 3–5) states that D is an element of the intersection of B and C, if (and only if) D is an element of B, and D is an element of C, i.e. $D \in B \cap C$ iff $D \in B$ and $D \in C$. Similar, the "subset_defn" axiom (lines 7–9) expresses the fact that $B \subseteq C$ iff $\forall D : D \in B \Rightarrow D \in C$. The conjecture (lines 11–13) represents equation (6).

On the iPod Nano, leanCoP solves this problem in 105 seconds (on standard hardware, leanCoP needs less than one second). The generated solution/proof consists of 38 proof steps. At CASC, the yearly world championship for automatic theorem provers, this problem was not solved by most of the provers including the two fastest theorem provers (for classical first-order logic) within the given time limit of 5 minutes.[4]

Analysis of Memory Usage

Table 2 shows (peak) RAM usage on the 1644 first-order problems of the TPTP library version 6.4.0 [22] that are proved by both, the core/iPod Nano prover of leanCoP 2.1 and the E prover [19] within 10 seconds on servers with Xeon 2.3 GHz CPUs (running Linux 2.6.32 and SWI-Prolog 5.0.10 Lite). On problem SYN915+1, both provers use the least amount of memory, on problem LCL490+1, E uses more than 60 times as much memory as leanCoP. On average, the E prover, which is one of the most powerful provers available, uses slightly more than twice the memory used by leanCoP.

memory / # problems	leanCoP	E
0 – 2,499 kbytes	1421	0
2,500 – 9,999 kbytes	215	1565
10,000 – 49,999 kbytes	8	66
50,000 – kbytes	0	13

	leanCoP	E
average memory	2,240	4,899
min. (SYN915+1)	1,588	2,800
max. (LCL490+1)	1,932	120,636
prover size (binary)	744	3,266

Table 2: Analysis of memory usage (in kbytes)

[4] See http://www.tptp.org/CASC/J6/; the Linux servers used in the competition were significantly more powerful than the iPod Nano, i.e., quad-core servers with 2.40 GHz CPUs and 48 GB of RAM.

6 Conclusion

Automated theorem proving is a key technology for automating formal reasoning. Over the last decades, the power of automatic theorem provers has improved significantly and they are used in industrial applications, such as the development of provably correct software. Most of the state-of-the-art provers are large and complex implementations and comprise ten thousands of lines of code, making these systems inflexible and difficult to maintain. More important, these provers are based on proof calculi that can generate thousands of new clauses during the proof search and, thus, require large amounts of memory. In contrast, connection calculi use significantly less memory and can be implemented in a compact and elegant way.

This paper presents a proof-of-concept that automatic reasoning techniques can run on small devices with very limited hardware resources. It uses a slightly adapted version of the leanCoP prover, a compact Prolog implementation of the connection calculus for first-order logic. The installation and evaluation of the prover on an iPod Nano shows that such an automatic reasoning tool can (on certain problems, see Sect. 5) outperform significantly more complex theorem provers running on much more powerful hardware.

Even though today's mobile devices have significant more memory and more powerful processors, up to the author's knowledge, no other first-order logic reasoning tool has been implemented on such a small mobile device so far. Just as the pocket calculator has automated the calculation in arithmetic on small portable devices, a *pocket reasoner* can automate logical reasoning on the move. Once, such a portable reasoning tool is powerful enough, it could help people to make *logical decisions* in everyday life.[5] Hence, future work include the porting of the presented reasoning tool to more powerful mobile devices and platforms.

Due to the compactness of the prover, it can easily be adapted to other logics and applications, such as *non-classical* logics [14]. Furthermore, the correctness of the code of the core prover can be checked much more easily.

Future work also includes adapting non-clausal reasoning approaches to small portable devices. For example, the *non-clausal* connection calculus is a generalization of the clausal connection calculus that does not require a translation into clausal form, but works directly on the structure of the input formula. Hence, the non-clausal connection calculus combines the advantages of more natural (non-clausal) sequent or tableau calculi with the efficient goal-oriented search of connection calculi. The compact nanoCoP implementation of this calculus does not only return more natural non-clausal proofs, but these proofs are also significantly shorter than the clausal proofs [11, 12].

References

[1] J. Barwise. An Introduction to First-Order Logic. In J. Barwise, ed., *Handbook of Mathematical Logic*, pages 5–46. North-Holland, Amsterdam, 1977.

[2] W. Bibel. Matings in matrices. *Communications of the ACM*, 26:844–852, 1983.

[3] W. Bibel. *Automated Theorem Proving*. Vieweg Verlag, Braunschweig, second edition, 1987.

[4] R.L. Constable et al. *Implementing Mathematics with the NuPRL proof development system*. Prentice–Hall, Englewood Cliffs/NJ, 1986.

[5] Observe that a *logical* decision is not necessarily a *good* decision, but it is a decision that takes possible (logical) consequences into account.

[5] G. Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935.

[6] R. Hähnle. Tableaux and Related Methods. In A. Robinson, A. Voronkov, eds., *Handbook of Automated Reasoning*, pages 100–178. Elsevier, Amsterdam, 2001.

[7] M. Kaufmann J. S. Moore. Some key research problems in automated theorem proving for hardware and software verification. *RACSAM*, 98(1):181–195, 2004.

[8] S.-J. Lee, D. Plaisted. Eliminating Duplicates with the Hyper-Linking Strategy. *Journal of Automated Reasoning*, 9:25–42, 1992.

[9] R. Letz, G. Stenz. Model Elimination and Connection Tableau Procedures. In A. Robinson, A. Voronkov, eds., *Handbook of Automated Reasoning*, pages 2015–2114. Elsevier, Amsterdam, 2001.

[10] J. Otten. Restricting backtracking in connection calculi. *AI Communications*, 23:159–182, 2010.

[11] J. Otten. nanoCoP: A Non-clausal Connection Prover. In N. Olivetti, A. Tiwari, eds., *IJCAR 2016*, LNAI 9706, pages 269–276. Springer, Heidelberg, 2016.

[12] J. Otten. nanoCoP: Natural Non-clausal Theorem Proving. In C. Sierra, ed., *International Joint Conference on Artificial Intelligence, IJCAI 2017*, Sister Conference Best Paper Track, pages 4924–4928. ijcai.org, 2017.

[13] J. Otten, W. Bibel. leanCoP: lean connection-based theorem proving. *Journal of Symbolic Computation*, 36:139–161, 2003.

[14] J. Otten, W. Bibel. Advances in connection-based automated theorem proving. In M. Hinchey et al., eds., *Provably Correct Systems*, NASA Monographs in Systems and Software Engineering, pages 211–241. Springer, Berlin, 2017.

[15] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[16] S. Ray. *Scalable Techniques for Formal Verification*. Springer, Heidelberg, 2010.

[17] J. A. Robinson, A. Voronkov. *Handbook of Automated Reasoning*. Elsevier, Amsterdam, 2001.

[18] J. M. Schumann. *Automated Theorem Proving in Software Engineering*. Springer, Heidelberg, 2001.

[19] S. Schulz. E – a brainiac theorem prover. *AI Communications*, 15:111–126, 2002.

[20] R. M. Smullyan. *First-Order Logic*. Springer, Heidelberg, 1968.

[21] M. Stickel. A Prolog technology theorem prover: implementation by an extended Prolog compiler. *Journal of Automated Reasoning*, 4:353–380, 1988.

[22] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure. *Journal of Automated Reasoning*, 59(4):483–502, 2017.

[23] A. Waaler. Connections in Nonclassical Logics. In A. Robinson, A. Voronkov, eds., *Handbook of Automated Reasoning*, pages 1487–1578. Elsevier, Amsterdam, 2001.