Preserving User Proofs Across Specification Changes

Jean-Christophe Filliâtre
CNRS

(joint work with François Bobot, Claude Marché,
Guillaume Melquiond, and Andrei Paskevich)

IFIP WG 1.9/2.15,
Saarbrucken, March 2013

Why3, a tool for deductive program verification

- logic = extension of first-order logic
  - with polymorphism, algebraic data types, inductive predicates
- programming language = ML-like
  - with mutable data structures but limited aliasing
- proofs = transformations + calls to external theorem provers
  - SMT solvers (Alt-Ergo, Z3, CVC3, Yices, etc.)
  - TPTP provers (SPASS, Eprover, Vampire, etc.)
  - proof assistants (Coq, PVS)

Example

Dijkstra's Dutch national flag
an array contains elements of the following enumerated type

```
type color = Blue | White | Red
```

sort it, in such a way we have the following final situation:

keep proofs up to date across

- user changes in specification and/or code
- evolution of the tool itself
  - e.g. change in WP calculus
- evolution of its standard library
- evolution of the environment
  - e.g. prover upgrades

Example

every night, we replay all the proofs for more than a hundred verified programs

in case of change

- we update the proof automatically when possible
- otherwise we signal the problem

What is a proof?

a forest, where each node is composed of

- a goal
  - = a logical context + a formula to be proved
- a list of proof attempts
  - a proof attempt = a call to an external prover and its result
- a list of transformations
  - each transformation produces a list of sub-trees

```
goal 1
  +- proof attempt "Alt-Ergo"
  +- proof attempt "Z3"
  +- transformation "split"
       +- goal 1.1
       +- goal 1.2
       |   +- proof attempt "CVC3"
       +- goal 1.3
goal 2
  +- transformation "induction"
       +- goal 2.1
       |    +- proof attempt "Alt-Ergo"
       +- goal 2.2
            +- proof attempt "Alt-Ergo"
```

Example

```
goal 1
  +- proof attempt "Alt-Ergo" (unknown 1.2s)
  +- proof attempt "Z3" (timeout 10s)
  +- transformation "split"
       +- goal 1.1
       +- goal 1.2
       |   +- proof attempt "CVC3" (valid 3s)
       +- goal 1.3
goal 2
  +- transformation "induction"
       +- goal 2.1
       |    +- proof attempt "Alt-Ergo" (valid 0.01s)
       +- goal 2.2
            +- proof attempt "Alt-Ergo" (valid 0.02s)
```

Example

```
goal 1
  +- proof attempt "Alt-Ergo" (unknown 1.2s)
  +- proof attempt "Z3" (timeout 10s)
  +- transformation "split"
       +- goal 1.1
       +- goal 1.2 verified
       |   +- proof attempt "CVC3" (valid 3s)
       +- goal 1.3
goal 2 verified
  +- transformation "induction" verified
       +- goal 2.1 verified
       |    +- proof attempt "Alt-Ergo" (valid 0.01s)
       +- goal 2.2 verified
            +- proof attempt "Alt-Ergo" (valid 0.02s)
```

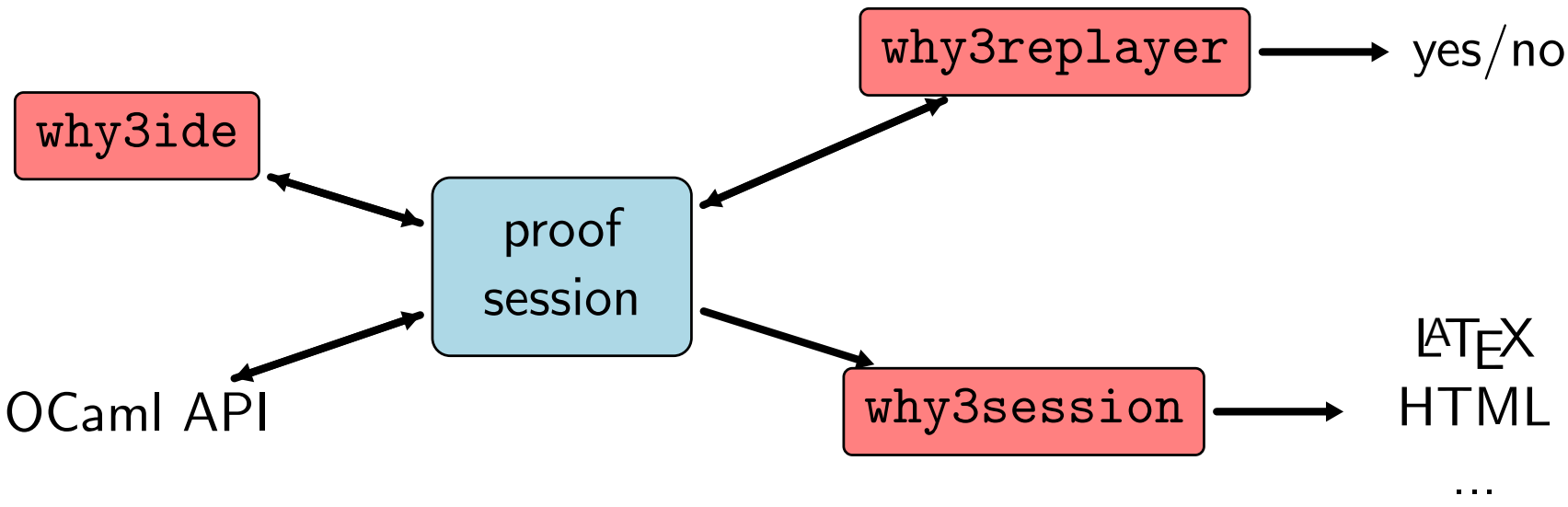
Proof Session

such a labeled forest is called a proof session

it is stored on disk as an XML file

several tools read or write it

Proof Session



Proof Session Update

the set of goals may change

- user modification of a goal statement
- modification of a goal context (e.g. additional hypotheses)
- modification of a program and/or its spec $\Rightarrow$ new VC
- modification of the tool / its library

problem: how to update the proof session?

Setting The Problem

input:

- a collection of (old) proofs
- a collection of (new) goals

output:

- a collection of (new) proofs

Algorithm

each new goal g is mapped

- either to a freshly created proof
  - with no proof attempt and no transformation
- or to an old proof P
  - each proof attempt of P becomes a proof attempt of g (it is obsolete and has to be run)
  - each transformation T of P is applied to g, and we proceed recursively with the old sub-proofs and the new sub-goals

Pairing

we are left with the sub-problem of pairing two lists of old and new goals

naturally, we first pair goals that are exactly the same
(we use MD5 checksums to do that)

we pair remaining goals using a heuristic measure of similarity
based on a notion of goal shape

Goal Shape

the shape of a goal is a character string

the similarity of two shapes is defined as the length of their common prefix

tries to match our intuition of logical similarity

- invariant by α-renaming
- conclusion is more important than hypotheses, e.g. the shape of $A \Rightarrow B$ is build from the shape of B first, and then the shape of A

Goal Shape

$$
\begin{aligned}
sh(n) &= \texttt{'c'} + n \\
sh(x) &= \texttt{'V'} + unique(x) \quad \text{if } x \text{ is a local variable} \\
&= \texttt{'V'} + x \quad \text{otherwise} \\
sh(\texttt{true}) &= \texttt{'t'} \\
sh(\texttt{false}) &= \texttt{'f'} \\
sh(f(t_1, \ldots, t_n)) &= \texttt{'a'} + f + sh(t_1) + \cdots + sh(t_n) \\
sh(\forall x : \tau.\, t) &= sh(t) + \texttt{'F'} \\
sh(t_1 \Rightarrow t_2) &= sh(t_2) + \texttt{'I'} + sh(t_1) \\
sh(\neg\ t) &= sh(t) + \texttt{'N'} \\
sh(\texttt{let } x = t_1 \texttt{ in } t_2) &= sh(t_2) + \texttt{'L'} + sh(t_1) \\
sh(\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3) &= \texttt{'i'} + sh(t_1) + sh(t_2) + sh(t_3)
\end{aligned}
$$

Example

sh(`forall x: int. f (f x) = x`) =

```
  a=afafV0V0F
```

Example

sh(`forall x: int. f (f x) = x`) =

```
a=afafV0V0F
```

sh(`forall n: int. 0 <= n` $\rightarrow$ `f (f n) = n`) =

```
a=afafV0V0Ia<=c0V0F
```

Example

$sh($`forall x: int. f (f x) = x`$) =$
`a=afafV0V0F`

$sh($`forall n: int. 0 <= n` $\rightarrow$ `f (f n) = n`$) =$
`a=afafV0V0Ia<=c0V0F`

$lcp($`a=afafV0V0F`, `a=afafV0V0Ia<=c0V0F`$) = 10$

Matching Algorithm

given N old shapes and M new shapes

$new \leftarrow$ new shapes
$old \leftarrow$ old shapes
while $new \neq \emptyset$ **and** $old \neq \emptyset$
  find o in old and n in new such that $lcp(o, n)$ is maximal
  pair o and n
  $old \leftarrow old - \{o\}$
  $new \leftarrow new - \{n\}$

Efficiency

(assuming $N = M$ for simplicity)

written naively, this algorithm runs in $O(N^3)$

if we sort shapes in lexicographic order, it becomes $O(N^2)$
(a pair (o, n) maximizing *lcp* is composed of consecutive elements)

if we store pairs of consecutive elements into a priority queue, it is even $O(N \log N)$

Efficiency

(assuming $N = M$ for simplicity)

written naively, this algorithm runs in $O(N^3)$

if we sort shapes in lexicographic order, it becomes $O(N^2)$
(a pair (o, n) maximizing *lcp* is composed of consecutive elements)

if we store pairs of consecutive elements into a priority queue, it is even $O(N \log N)$

$$O(L(N + M) \log(N + M))$$

Empirical Evaluation

we observe very good results whenever

- we only change the standard library
- we introduce additional hypotheses, e.g. a new precondition

yet from time to time a proof is incorrectly matched or lost
(then for Coq or PVS we have to manually recover the former proof script)

Discussion

better shapes

- unambiguous
- insensitive to hypotheses swapping
- including types

a different matching algorithm

- string edit distance (Levenshtein)
- tree edit distance (e.g. Zhang & Shasha, 1989)
- minimizing sum of distances

(Not so) Related Work

systems built on top of a single, automated theorem prover are fine with redoing proofs from scratch (e.g. Dafny, VeriFast)

other systems benefit from complete proofs (e.g. KIV, KeY) and may use that information for smarter proof re-use