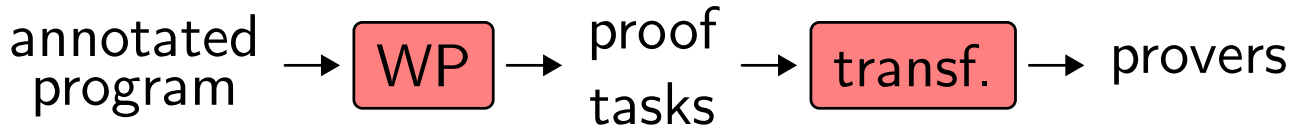
Memo Tables

Jean-Christophe Filliâtre
CNRS

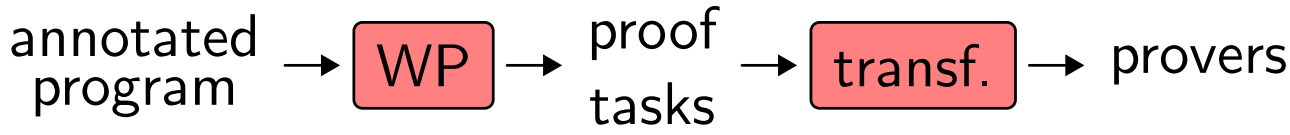
joint work with Francois Bobot and Andrei Paskevich
ProVal team, Orsay, France

IFIP WG 2.8 Functional Programming
March 2011

Context: Deductive Program Verification

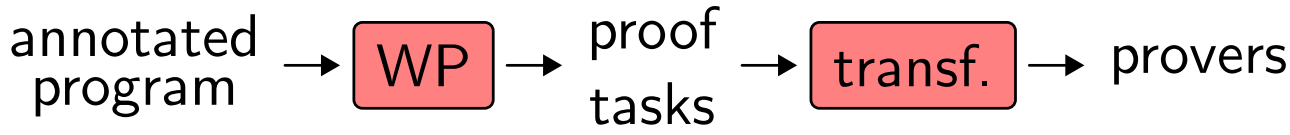


Context: Deductive Program Verification



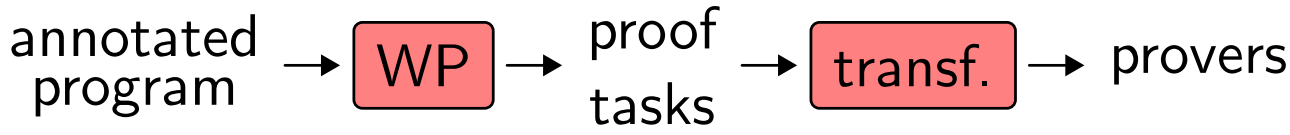
- C / Java programs
- ML programs
- pre/post-conditions
- invariants

Context: Deductive Program Verification



- C / Java programs
- ML programs
- pre/post-conditions
- invariants
- polymorphic first-order logic
- algebraic data types
- inductive predicates

Context: Deductive Program Verification



- C / Java programs
- ML programs
- pre/post-conditions
- invariants

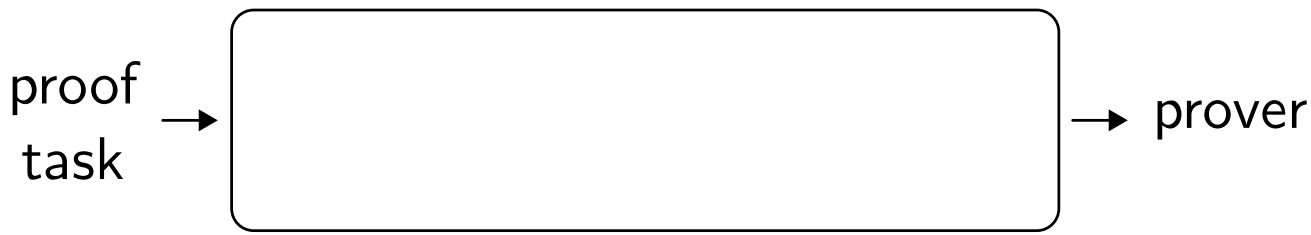
- polymorphic first-order logic
- algebraic data types
- inductive predicates

- untyped, many-sorted, etc.
- few or no algebraic data types
- some built-in theories (arithmetic, arrays, etc.)

Context: Deductive Program Verification

Why3: new implementation started one year ago

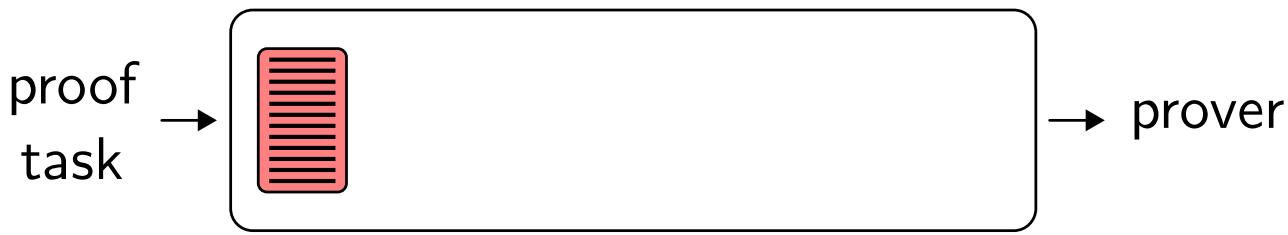
key notion: transformation



Context: Deductive Program Verification

Why3: new implementation started one year ago

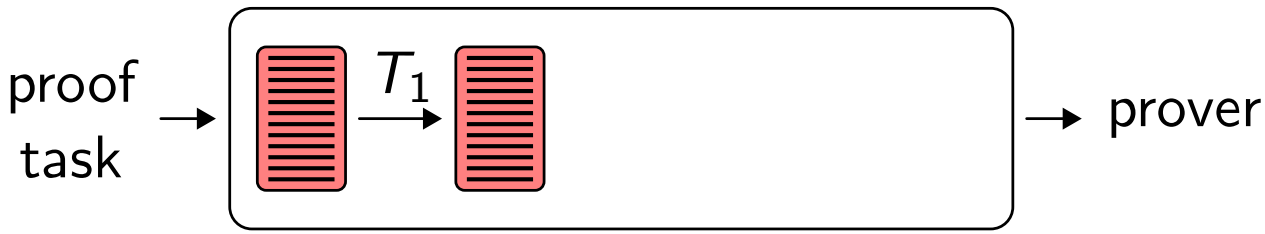
key notion: transformation



Context: Deductive Program Verification

Why3: new implementation started one year ago

key notion: transformation



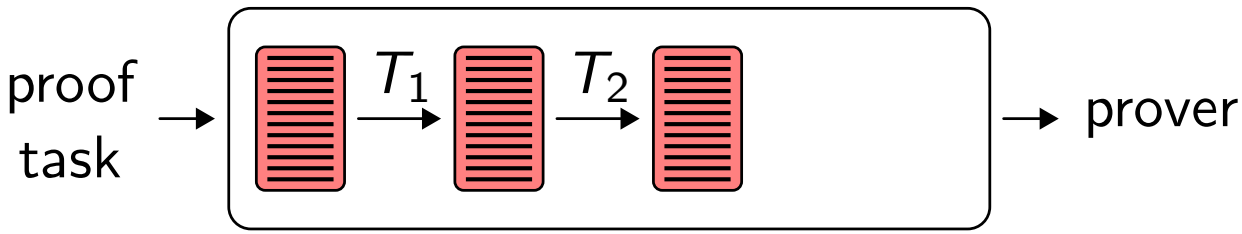
example

- $T_1 =$ inlining of simple definitions

Context: Deductive Program Verification

Why3: new implementation started one year ago

key notion: transformation



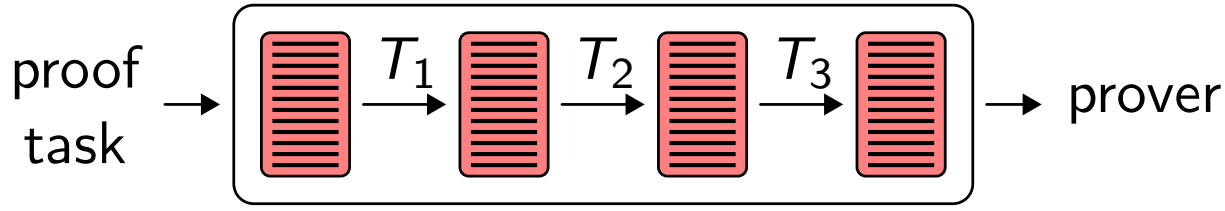
example

- T_1 = inlining of simple definitions
- T_2 = elimination of algebraic types

Context: Deductive Program Verification

Why3: new implementation started one year ago

key notion: transformation



example

- T_1 = inlining of simple definitions
- T_2 = elimination of algebraic types
- T_3 = encoding of polymorphism

Efficiency Concerns

to save space, we do

- hash-consing of terms, formulas and task prefixes

to save time, we do

- memoization of transformation functions

Memo Tables

there are millions of task elements, thousands of transformations

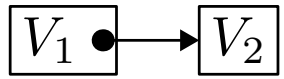
some are long-lived, others short-lived

we need efficient memo tables to avoid memory leaks

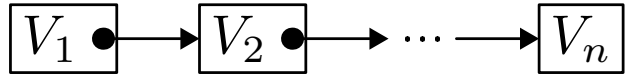
The Problem

Terminology

- a value can point to another value



- a value is reachable from another value



- a set of values called roots
  any value not reachable from a root can be reclaimed

The Problem

some values are called keys, some values are called tables

to a key K and a table T we can assign an arbitrary value V, written $T : K \mapsto V$

given an existing binding $T : K \mapsto V$, we can remove it, undoing the corresponding assignment

The Problem: Requirements

given a binding $T : K \mapsto V$

as long as K and T are both reachable, then V is reachable too (and can be obtained from K and T)

The Problem: Requirements

if K is reachable, then it is still reachable when all bindings $T : K \mapsto V$ are removed

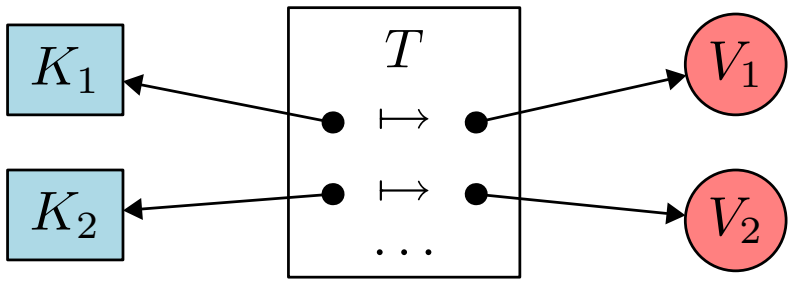
if T is reachable, then it is still reachable when all bindings $T : K \mapsto V$ are removed

if V is reachable, then it is still reachable when all bindings $T : K \mapsto V$ with K or T unreachable are removed

Some (Partial) Solutions

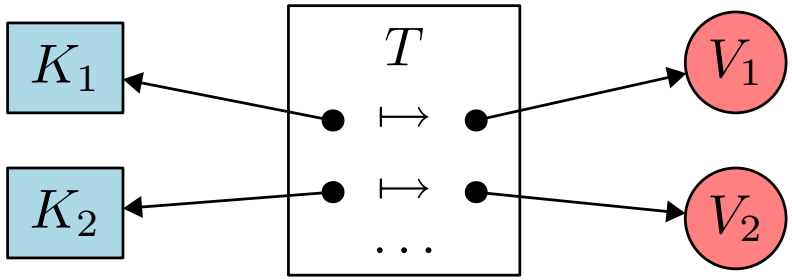
Naive Solution

T is a traditional dictionary data structure
(hash table, balanced tree, etc.)



Naive Solution

T is a traditional dictionary data structure
(hash table, balanced tree, etc.)



obvious drawback

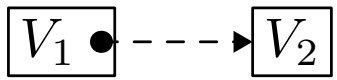
T reachable $\Rightarrow$ all keys and values bound in T are also reachable

conclusion

T should not hold pointers to keys

New Tool: Weak Pointers

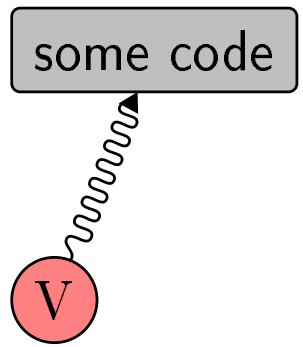
a value can weakly point to another value, depicted



a value not yet reclaimed can be accessed via a weak pointer

New Tool: Finalizers

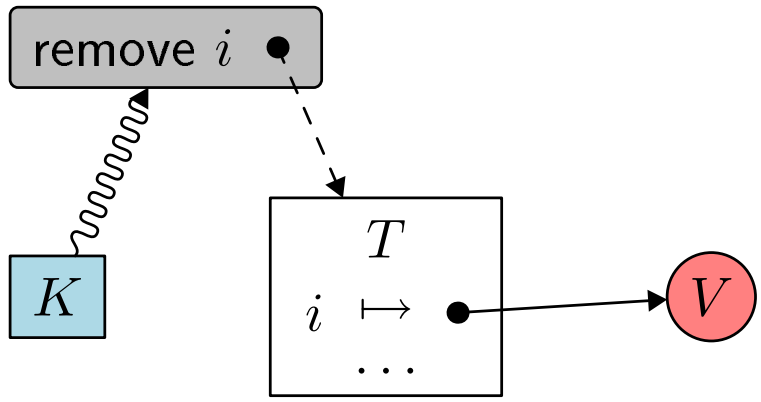
one or several finalizers can be attached to a value



a finalizer is a closure which is executed whenever the corresponding value is going to be reclaimed

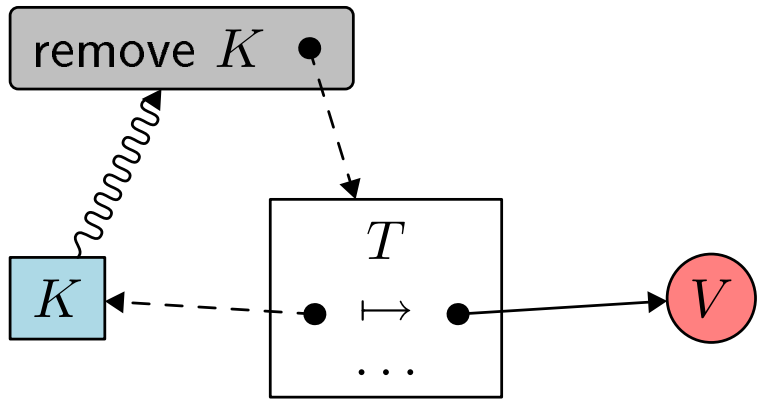
A Better Solution?

K is not used directly as index in T
but a unique tag i is used instead



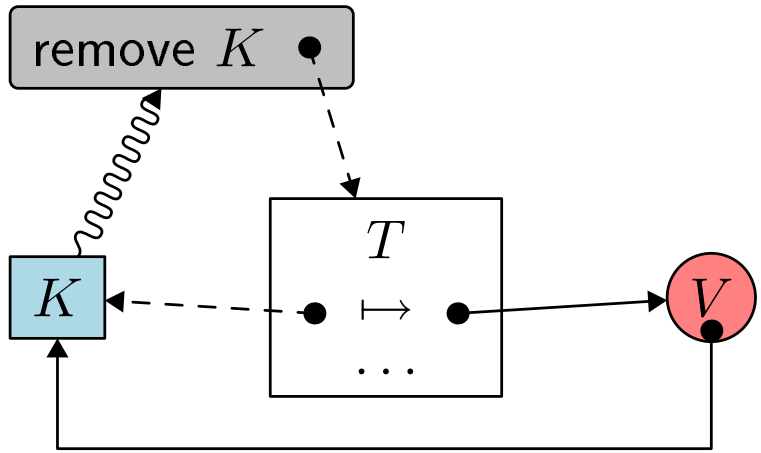
A Better Solution?

K is not used directly as index in T



A Better Solution?

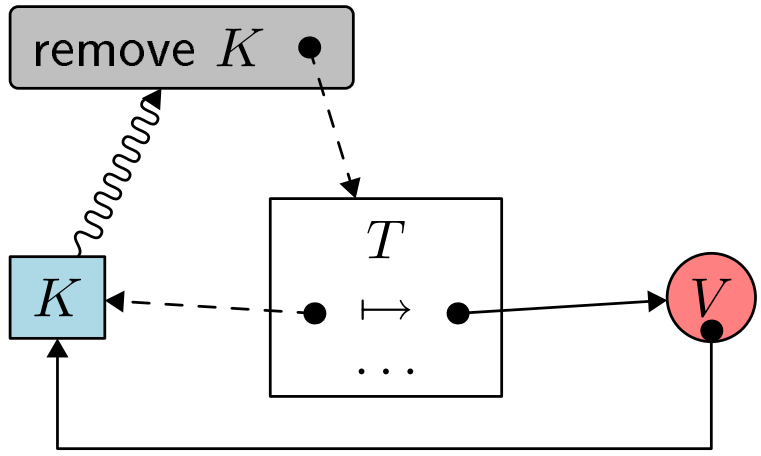
it seems to be a good solution...
but a key can be reachable from a value (*e.g.* $V = K$)



preventing K from being reclaimed

A Better Solution?

it seems to be a good solution...
but a key can be reachable from a value (*e.g.* $V = K$)



preventing K from being reclaimed

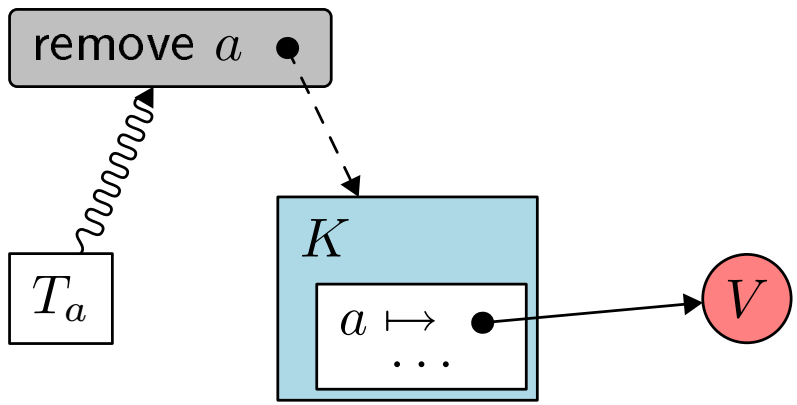
conclusion

T should not hold pointers to values either

A Better Solution!

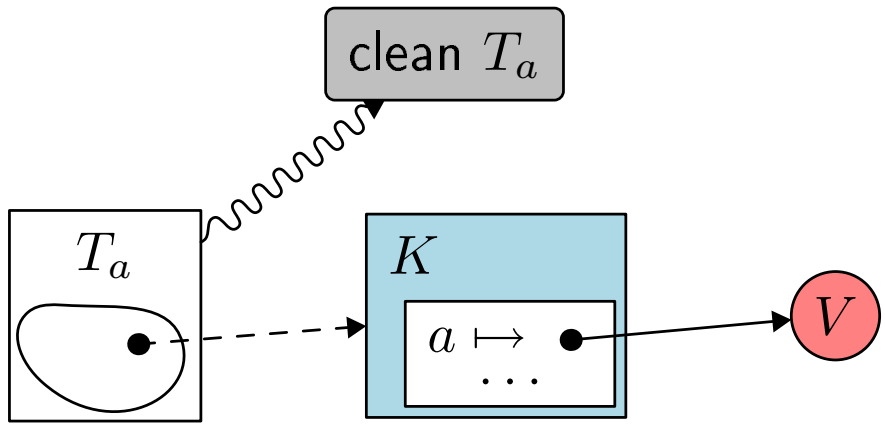
we cannot stock bindings inside tables

$\Rightarrow$ let us keep them in keys instead



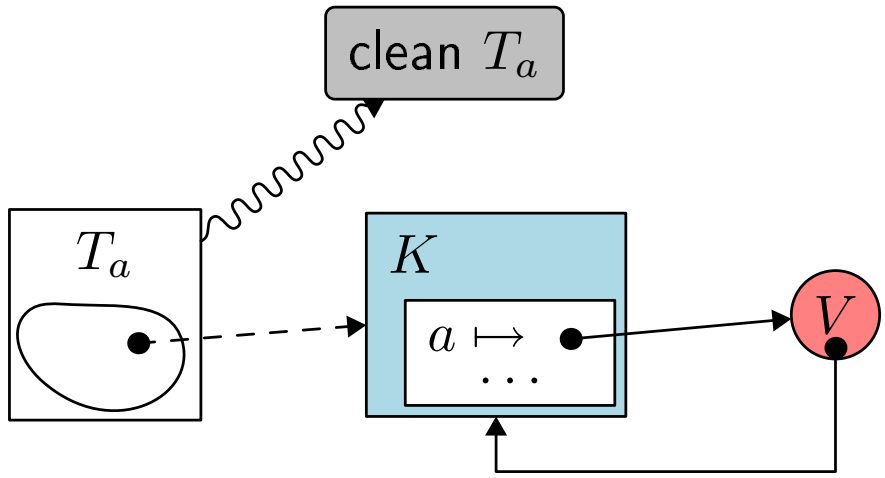
A Better Solution!

improvement: only one finalizer instead of one per key



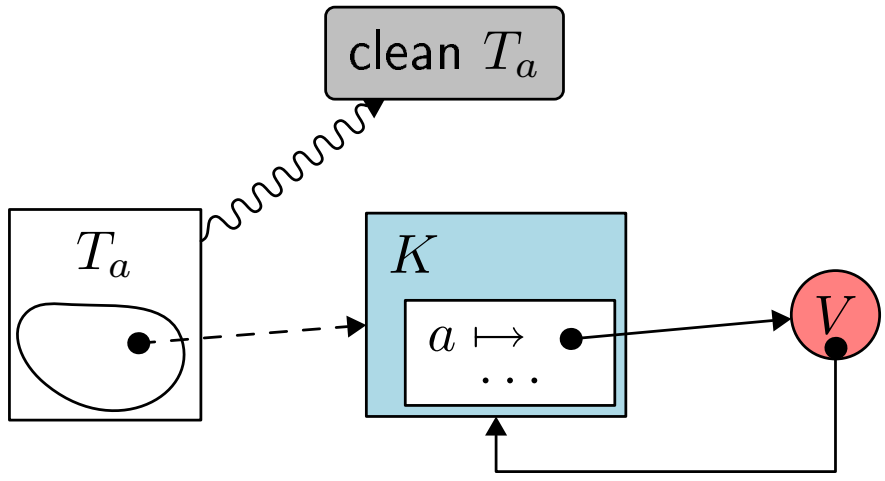
A Better Solution!

K reachable from V is not a problem anymore



A Better Solution!

K reachable from V is not a problem anymore

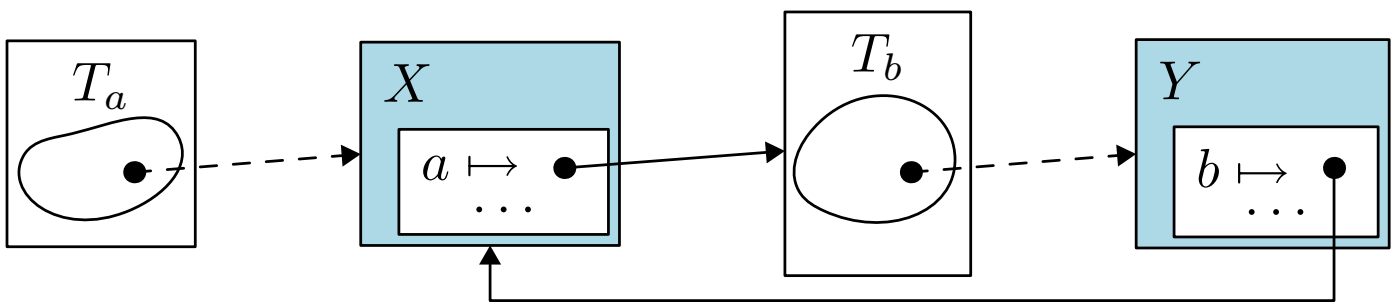


(note: you can implement a similar solution in Haskell using `System.Mem.Weak`)

Symmetry

of course, the roles of K and T being symmetric,
if T is reachable from V the "cycle issue" is still there

example: we want to memoize the **K** combinator $\mathbf{K}(X, Y) = X$
we first memoize the partial application to X, the result being another memoization table



Symmetry

the approach is viable if we can guarantee that the first argument always lives longer than the second one

fortunately, this is indeed the case in our framework

Implementation

implemented as an Ocaml library

```
type tag

type α tagged = private {
  node : α;
  tag  : tag;
}

val create : α → α tagged

val memoize : (α tagged → α) → (α tagged → α)
```

Implementation

implemented as an Ocaml library

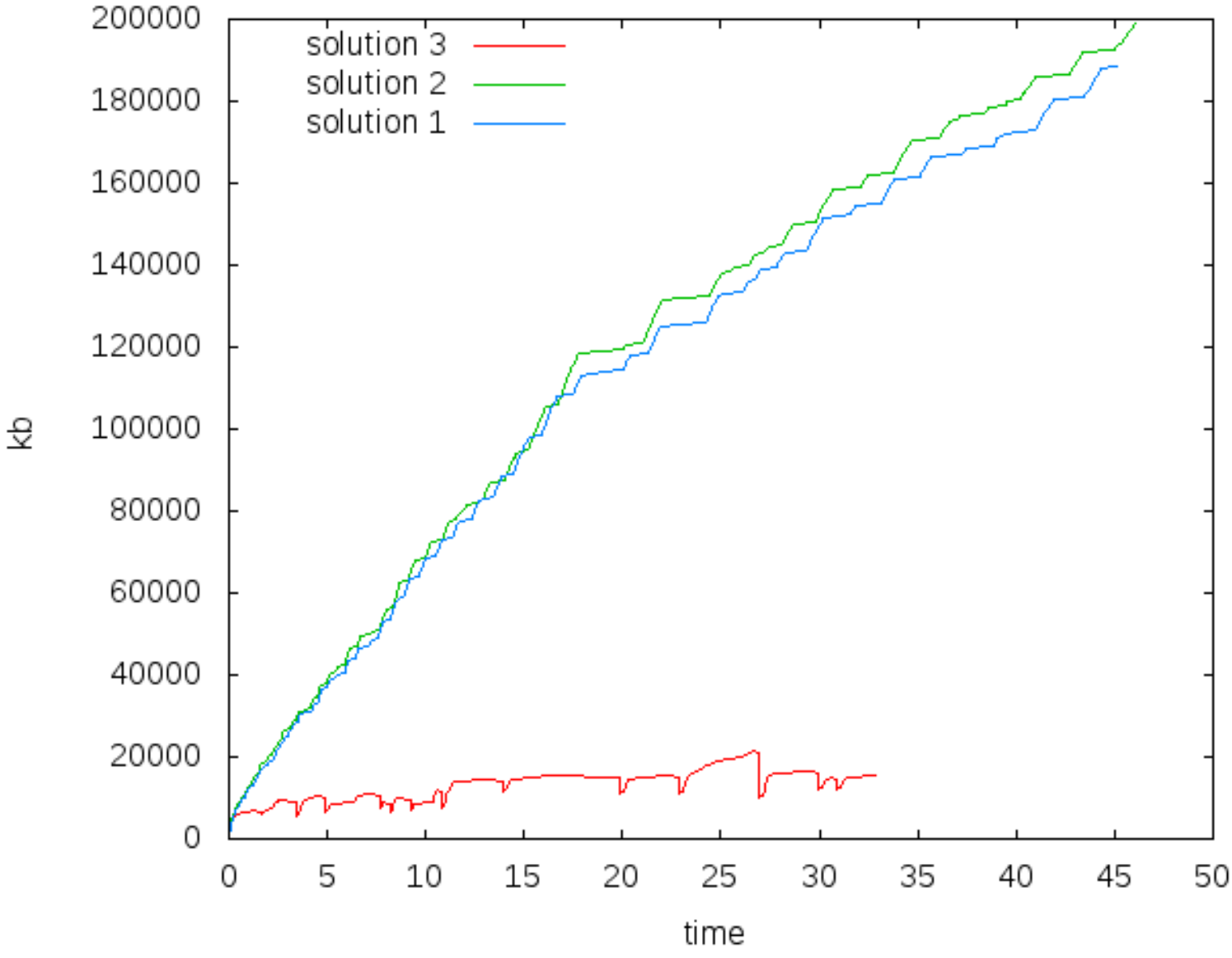
```
type tag

val create : unit → tag

module Memo
  (Key : sig type t
             val  tag : t → tag end) :
sig
  val memoize : (Key.t → α) → (Key.t → α)
end
```

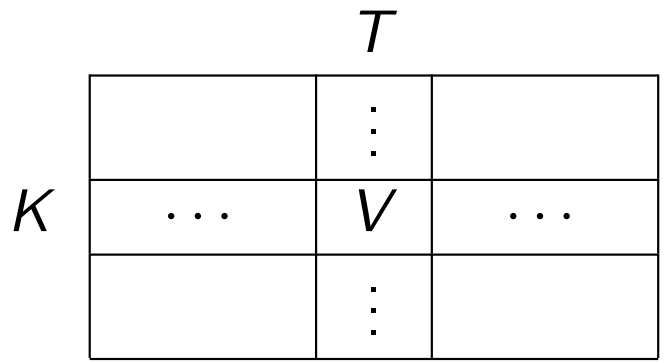
Benchmarks

1,448 proof tasks translated to SMT-lib format and printed in files



Discussion

we can rephrase the problem in terms of a single, immortal table with several keys



where V is removed as soon as K or T is reclaimed

can we propose a new notion of weak pointer for that purpose?