Science of Computer Programming 164 (2018) 49–65

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

A Coq library for internal verification of running-times

Jay McCarthy [a,*], Burke Fetscher [b], Max S. New [b], Daniel Feltey [b], Robert Bruce Findler [b]

[a] University of Massachusetts Lowell, United States
[b] Northwestern University, United States

ARTICLE INFO

Article history:
Received 16 June 2016
Received in revised form 27 April 2017
Accepted 8 May 2017
Available online 19 May 2017

Keywords:
Mechanized proofs
Running-time
Complexity
Coq

ABSTRACT

This paper presents a Coq library that lifts an abstract yet precise notion of running-time into the type of a function. Our library is based on a monad that counts abstract steps. The monad's computational content, however, is simply that of the identity monad so programs written in our monad (that recur on the natural structure of their arguments) extract into idiomatic OCaml code.

We evaluated the expressiveness of the library by proving that red-black tree insertion and search, merge sort, insertion sort, various Fibonacci number implementations, iterated list insertion, various BigNum operations, and Okasaki's Braun Tree algorithms all have their expected running times.

1. Introduction

For some programs, proving that they have correct input–output behavior is only part of the story. As Crosby and Wallach [10] observed, incorrect performance characteristics can lead to security vulnerabilities. Indeed, some programs and algorithms are valuable precisely because of their performance characteristics. For example, merge sort is preferable to insertion sort only because of its improved running time. Unfortunately, defining functions in Coq or other theorem proving systems does not provide enough information in the types to state these intensional properties.

Our work provides a monad (implemented as a library in Coq) that enables us to include abstract running times in types. We use this library to prove that several important algorithms have their expected running times.

The monad in our work has similar goals to the one in Danielsson's [11], but with two benefits. First, it allows programmers to write idiomatic code without embedding invariants in data types, so we can reason about a wider variety of programs. Second, and more significantly, our monad adds no complexity computations to the extracted OCaml code, so the verification imposes no run-time overhead. We elaborate these details and differences throughout the paper and, in particular, in section 9.

The rest of the paper is structured as follows. In section 2, we give an overview of how the library works and the style of proofs we support. In section 3, we discuss the cost model our proofs deal with. In section 4, we explain the extraction of our programs to OCaml. In these first three sections, we use a consistent example that is introduced in section 2. Following this preamble, section 5 walks through the definition and design of the monad itself. Section 6 describes the results of our case study, wherein we proved properties of a variety of different functions. Section 7 and section 8 discuss accounting

* Corresponding author.
E-mail addresses: jay.mccarthy@gmail.com (J. McCarthy), burke.fetscher@eecs.northwestern.edu (B. Fetscher), max.new@eecs.northwestern.edu (M.S. New), daniel.feltey@eecs.northwestern.edu (D. Feltey), robby@eecs.northwestern.edu (R.B. Findler).

http://dx.doi.org/10.1016/j.scico.2017.05.001
0167-6423/

```
Program Fixpoint insert {A:Set} (i:A) (b:@bin_tree A)
: {! res !:! @bin_tree A !<! c !>!
     (forall n, Braun b n -> (Braun res (n+1) /\ c = fl_log n + 1)) !} :=
  match b with
    | bt_mt         => += 1; <== (bt_node i bt_mt bt_mt)
    | bt_node j s t => t' <- insert j t;
                       += 1; <== (bt_node i t' s)
  end.
```

Fig. 1. Braun tree insertion.

for the running time of various language primitives. Finally, section 9 provides a detailed account of our relation to similar projects. Our source code and other supplementary material is available at https://github.com/rfindler/395-2013.

Extended material: Compared to the conference proceedings version of this paper [19], this version contains more elaborate and detailed figures and proofs throughout, as well as an extended discussion of language primitive runtimes in section 7.

2. Overview of our library

The core of our library is a monad that, as part of its types, tracks the running time of functions. To use the library, programs must be explicitly written using the usual return and bind monadic operations. In return, the result type of a function can use not only the argument values to give it a very precise specification, but also an abstract step count describing how many primitive operations (function calls, pattern matches, variable references etc.) that the function executes.

To give a sense of how code using our library looks, we start with a definition of Braun trees [5] and their insertion function, where the contributions to the running time are explicitly declared as part of the body of the function. In the next section, we make the running times implicit (and thus not trusted or spoofable).

Braun trees, which provide for efficient growable vectors, are a form of balanced binary trees where the balance condition allows only a single shape of trees for a given size. Specifically, for each interior node, either the two children are exactly the same size or the left child's size is one larger than the right child's size.

Because this invariant is so strong, explicit balance information is not needed in the data structure that represents Braun trees; we can use a simple binary tree definition.

```
Inductive bin_tree {A:Set} : Set :=
| bt_mt   : bin_tree
| bt_node : A -> bin_tree -> bin_tree -> bin_tree.
```

To be able to state facts about Braun trees, however, we need the inductive `Braun` to specify which binary trees are Braun trees (at a given size `n`).[1]

```
Inductive Braun {A:Set} : (@bin_tree A) -> nat -> Prop :=
| B_mt   : Braun bt_mt 0
| B_node : forall (x:A) s s_size t t_size,
  t_size <= s_size <= t_size+1 ->
  Braun s s_size -> Braun t t_size ->
  Braun (bt_node x s t) (s_size+t_size+1).
```

This says that the empty binary tree is a Braun tree of size `0`, and that if two numbers `s_size`, `t_size` are the sizes of two Braun trees `s` and `t`, and if `t_size <= s_size <= t_size+1`, then combining `s` and `t` into a single tree produces a Braun tree of size `s_size+t_size+1`.

Fig. 1 shows the insertion function. Let us dig into this function, one line at a time. It accepts an object `i` (of type `A`) to insert into the Braun tree `b`. Its result type uses a special notation:

```
{! «result id» !:! «simple type» !<! «cost id» !>! «property» !}
```

where the braces, exclamation marks, colons, less than, and greater than are all fixed parts of the syntax and the portions enclosed in « » are filled in based on the particulars of the insert function. In this case, it is saying that `insert` returns a binary tree and, if the input is a Braun tree of size `n`, then the result is a Braun tree of size `n+1` and the function takes `fl_log n + 1` steps of computation (where `fl_log` computes the floor of the base 2 logarithm and is defined as zero at zero).

These new `{! ... !}` types are the types of computations in the monad. The monad tracks the running time and verifies the correctness property of the function.

[1] The `@` in `bin_tree` is to specify the implicit type argument.