Verified Analysis of Functional Data Structures with Isabelle/HOL

Tobias Nipkow

Fakultät für Informatik
Technische Universität München

2017-7-31
Chapter 1

Introduction
1 Motivation

2 Time

3 Binary Trees
1 Motivation

2 Time

3 Binary Trees
General aims

- Verification of correctness and complexity of functional data structures
- Algebraic proofs about functions and their execution time
- Verified in an interactive theorem prover (Isabelle/HOL)

Complexity analysis should be as algebraic as proving

\[ \binom{r}{m} \binom{m}{k} = \binom{r}{k} \binom{r-k}{m-k} \]

Part of a project on Verified Algorithm Analysis
This course

- An introduction to the basic methods
- A whirlwind tour of verified functional data structures (and some proofs)
- Specifically: “Classical” search trees and priority queues
- First functional versions:
- Now: verified algebraic proofs
  See src/HOL/Data_Structures in the Isabelle distribution or online
1 Motivation

2 Time

3 Binary Trees
2 Time

A Lightweight Approach
A Monadic Approach
Amortized Analysis
Principle: Count function calls

For every function \( f :: \tau_1 \Rightarrow \ldots \Rightarrow \tau_n \Rightarrow \tau \)
define a *timing function* \( t_f :: \tau_1 \Rightarrow \ldots \Rightarrow \tau_n \Rightarrow \text{nat} \):

Translation of defining equations:

\[
\begin{align*}
ed & \leadsto e' \\
\quad \quad f \ p_1 \ldots p_n = e & \leadsto t_\ f \ p_1 \ldots p_n = e' + 1
\end{align*}
\]

Translation of expressions:

\[
\begin{align*}
s_1 & \leadsto t_1 \quad \ldots \quad s_k & \leadsto t_k \\
\quad \quad g \ s_1 \ldots s_k & \leadsto t_1 + \cdots + t_k + t_\ g \ s_1 \ldots s_k
\end{align*}
\]

- Variable \( \leadsto 0 \), Constant \( \leadsto 0 \)
- Constructor calls and primitive operations on \( \text{bool} \) and numbers cost 1
Example

\[\text{app} \; [] \; ys = ys\]

\[\Rightarrow\]

\[\text{t_app} \; [] \; ys = 0 + 1\]

\[\text{app} \; (x\#xs) \; ys = x \# \; \text{app} \; xs \; ys\]

\[\Rightarrow\]

\[\text{t_app} \; (x\#xs) \; ys = 0 + (0 + 0 + \text{t_app} \; xs \; ys) + 1 + 1\]
A compact formulation of \( e \leadsto t \)

\( t \) is the sum of all \( t_g s_1 \ldots s_k \)
such that \( g s_1 \ldots s_n \) is a subterm of \( e \)

If \( g \) is

- a constructor or
- a predefined function on \( \text{bool} \) or numbers

then \( t_g \ldots = 1 \).
So far we model a call-by-value semantics

Conditionals and case expressions are evaluated lazily.

Translation:

\[
\begin{align*}
    b & \rightsquigarrow t \\
    s_1 & \rightsquigarrow t_1 \\
    s_2 & \rightsquigarrow t_2 \\
\end{align*}
\]

\[
\text{if } b \text{ then } s_1 \text{ else } s_2 \rightsquigarrow t + (\text{if } b \text{ then } t_1 \text{ else } t_2)
\]

Similarly for case
$O(\cdot)$ is enough

$\implies$ Reduce all additive constants to 1

Example

\[
t_{\text{app}} (x\#xs) \ ys = t_{\text{app}} \ xs \ ys + 1
\]
2 Time

A Lightweight Approach
A Monadic Approach
Amortized Analysis
• Start with the definition of a function $f_{tm}$ that computes both the result and the running time simultaneously
• Define $f_{tm}$ via a monad that counts function calls behind the scene
• Derive $f$ and $t_f$ from $f_{tm}$
The time monad

datatype 'a tm = TM 'a nat

val (TM v n) = v

time (TM v n) = n

The programmer must only use:

Bind:

\[ Tm\ u\ m \gg f = (let\ Tm\ v\ n = f\ u\ in\ Tm\ v\ (m+n)) \]

Return (counts function call):

\[ ret\ v = TM\ v\ 1 \]

Use do-notation rather than \[\gg\]
Simple example

\[
\begin{align*}
app_{\text{tm}} &:: \ 'a \ \text{list} \Rightarrow \ 'a \ \text{list} \Rightarrow \ 'a \ \text{list} \ \text{tm} \\
app_{\text{tm}} \ [\] \ y_\text{s} & = \ \text{ret} \ y_\text{s} \\
app_{\text{tm}} \ (x \# \ x_\text{s}) \ y_\text{s} & = \ \text{do} \ \{ \\
& \quad \ z_\text{s} \gets \ app_{\text{tm}} \ x_\text{s} \ y_\text{s}; \\
& \quad \ \text{ret} \ (x \# \ z_\text{s}) \\
\}
\end{align*}
\]
Simple example

Define

\[ app \; xs \; ys = val \; (app\_tm \; xs \; ys) \]
\[ t\_app \; xs \; ys = time \; (app\_tm \; xs \; ys) \]

Prove automatically:

\[ app \; [] \; ys = [] \]
\[ app \; (x \# \; xs) \; ys = (let \; zs = app \; xs \; ys \; in \; x \# \; zs) \]

Prove with (some) insight:

\[ t\_app \; xs \; ys = length \; xs + 1 \]
2 Time

A Lightweight Approach
A Monadic Approach
Amortized Analysis
Recall

Amortized complexity = average complexity of single operation in a sequence of operations, in the worst case

Example

Consider a $k$-bit binary counter. An individual increment has complexity $O(k)$. In a sequence of increments starting from 0, each increment has amortized constant complexity.
Formalization

Implementation: Data structure of type $\tau$ with

- Operation(s) $f :: \tau \Rightarrow \tau$
  (may have additional parameters)
- Initial value: $\text{init} :: \tau$
  (function “empty”)

Claim: Operation $f$ has a certain amortized complexity of at most $a_f :: \tau \Rightarrow \text{num}$.
(Typically $a_f$ is constant or logarithmic)
Potential method

Find a potential function $\Phi : \tau \Rightarrow \text{num}$ and prove

- $\Phi \init = 0$
- $\Phi \ s \geq 0$
- for each operation $f$:
  \[ t_f \ s + \Phi(f \ s) - \Phi \ s \leq a_f \ s \]

amortized cost

What does this imply?
Amortized and real cost

A sequence of operations $f_1, \ldots, f_n$ induces a sequence of states:

$$ s_0 = init, \quad s_1 = f_1 s_0, \quad \ldots, \quad s_n = f_n s_{n-1} $$

The real cost of performing $f_1, \ldots, f_n$ is upper-bounded by amortized costs:

$$ \sum_{i=1}^{n} t_{-f_i} s_{i-1} \leq \sum_{i=1}^{n} a_{-f_i} s_{i-1} $$
Warning

Amortized analysis is only correct for single threaded uses of a data structure.

Single threaded = no value is used more than once

Otherwise:

\[
\begin{align*}
\text{let} & \quad \text{counter} = 0; \\
\text{bad} & \quad = \text{increment} \text{ counter} \ 2^n - 1 \ 	ext{times}; \\
_ & \quad = \text{incr bad}; \\
_ & \quad = \text{incr bad}; \\
_ & \quad = \text{incr bad}; \\
\vdots
\end{align*}
\]
1 Motivation

2 Time

3 Binary Trees
Binary trees

datatype 'a tree = Leaf | Node ('a tree) 'a ('a tree)

Abbreviations:
\[
\langle \rangle \equiv Leaf \\
\langle l, a, r \rangle \equiv Node l a r
\]

Most of the time: tree = binary tree
Tree traversal

\[
i \text{inorder :: } 'a \text{ tree } \Rightarrow 'a \text{ list}
\]

\[
i \text{inorder } \langle \rangle = [\]
\]

\[
i \text{inorder } \langle l, x, r \rangle = \text{inorder } l @ [x] @ \text{inorder } r
\]
Size

Number of nodes:

\[
size :: 'a \text{ tree} \Rightarrow \text{nat}
\]

\[
\| \langle \rangle \| = 0
\]

\[
\| \langle l, -, r \rangle \| = \| l \| + \| r \| + 1
\]

Number of leaves:

\[
size1 :: 'a \text{ tree} \Rightarrow \text{nat}
\]

\[
| t |_1 = | t | + 1
\]
height :: 'a tree ⇒ nat

\[ h(\langle \rangle) = 0 \]
\[ h(\langle l, _, r \rangle) = \max (h(l)) \cdot (h(r)) + 1 \]

Lemma \(|t|_1 \leq 2^{h(t)}\)