The Resource Bound Problem for Imperative Languages

Quentin Carbonneaux

April 21, 2015
Suppose we have \( n \) consecutive stack operations.

\[
push() ; \\
popall() ; \\
push() ; \\
push() ; \\
\vdots
\]

popall costs \( s \), push costs 1.

The stack starts empty.
We can rephrase the setting using non-determinism and a loop.

```c
while n > 0
    if *
        push();
    else
        popall();
    n = n - 1;
```
We can rephrase the setting using non-determinism and a loop.

```java
while n > 0
    if *
        push();
    else
        popall();
    n = n - 1;
```

What is the worst-case?
Answer: $2n$

The number of elements pushed cannot be bigger than $n$. Thus,

- the combined cost for all the pops is $\leq n$,
- and the combined cost for all pushes is $\leq n$.

Hence $2n$. 
This is nice reasoning, but informal. It’s unclear how to scale this to real programs. We need

- Formal Cost Semantics
- Formal Logical Reasoning
- Automation

My work brings some answers to these 3 points.
Potential Method and Quantitative Logic
A Good Start: Tarjan’s Idea

Tarjan proposes to find a potential function.

- $\sigma, \sigma'$ are program states.
- $C_l$ is the cost of one loop iteration.
- $\Phi : \text{State} \rightarrow \mathbb{Z}$ is a potential function iff

$$\Phi(\sigma) \geq \begin{cases} 0 & \text{if } \sigma(n) \leq 0 \\ C_l + \Phi(\sigma') & \text{if } \sigma(n) > 0 \end{cases}.$$
Why?

\[
\begin{align*}
\Phi(\sigma_0) & \geq C_l + \Phi(\sigma_1) \\
\Phi(\sigma_1) & \geq C_l + \Phi(\sigma_2) \\
\Phi(\sigma_2) & \geq C_l + \Phi(\sigma_3) \\
\vdots & \\
\Phi(\sigma_f) & \geq 0
\end{align*}
\]

\[\sum C_l \text{ is the total program cost.}\]
Claim: Taking $\Phi = 2n + s$ Works

push case ($C_l = 1$):

$$\Phi(n, s) = 2n + s = 1 + 2(n - 1) + s + 1 \\
\geq C_l + \Phi(n - 1, s + 1).$$

popall case ($C_l = s$):

$$\Phi(n, s) = 2n + s = s + 2(n - 1) + 1 \\
\geq C_l + \Phi(n - 1, 0).$$

base case: $\Phi(0, s) = s \geq 0$
What did we prove?

\[ \Phi(n, 0) = 2n \] bounds the number of stack ops.
What did we prove?

Φ(n, 0) = 2n bounds the number of stack ops.

We proved more! If the stack is not initially empty, Φ(n, s) is a correct bound. We had to introduce s in Φ for the induction.
Using triples, we get compositional resource bound proofs:

Write $\{\Phi\} S \{\Phi'\}$ to mean $\Phi(\sigma) \geq C_S + \Phi'(\sigma')$. 
Potential Function $\implies$ Compositional

Using triples, we get compositional resource bound proofs:

Write $\{\Phi\}S\{\Phi'\}$ to mean $\Phi(\sigma) \geq C_S + \Phi'(\sigma')$.

If $\{\Phi\}S_1\{\Phi'\}$ and $\{\Phi'\}S_2\{\Phi''\}$ then

$$\{\Phi\}S_1;S_2\{\Phi''\}.$$ 

This is telescoping, because $CS_1;S_2 = CS_1 + CS_2$. 
\[
\frac{\{\Phi\} S_1 \{\Phi'\} \quad \{\Phi'\} S_2 \{\Phi''\}}{\{\Phi\} S_1 ; S_2 \{\Phi''\}} \quad (Q:SEQ)
\]
\[
\frac{\{X \land \Phi\} S \{\Phi\}}{\{\Phi\} \text{while } X \text{ do } S \{-X \land \Phi\}} \quad (Q:\text{LOOP})
\]
Combining Logic and Potential

<table>
<thead>
<tr>
<th>Logic Assertions</th>
<th>Potential Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>State $\rightarrow \mathbb{B}$</td>
<td>State $\rightarrow \mathbb{Q}_0^+ \cup {\infty}$</td>
</tr>
<tr>
<td>$\top$</td>
<td>0, 1, 1.5, 2, \ldots</td>
</tr>
<tr>
<td>$\bot$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$\land$</td>
<td>$+$</td>
</tr>
<tr>
<td>$\lor$</td>
<td>$\min$</td>
</tr>
</tbody>
</table>

Example: $\top \lor \bot = \top$ translates to $\min(0, \infty) = 0$. 
Complete Logic for Clight

\[
\begin{align*}
\Delta; B; R \vdash_L \{ Q \} \text{ skip } \{ Q \} & \quad \text{(L: Skip)} \\
n < 0 \implies Q \geq 0 & \quad \text{(L: Tick)} \\
\Delta; B; R \vdash_L \{ Q \} \text{ tick } (n) \{ Q - n \} & \quad \text{(L: Tick)} \\
\Delta; B; R \vdash_L \{ e \} \sigma \implies Q \text{ assert } e \{ Q \} & \quad \text{(L: Assert)} \\
I \geq Q & \quad \Delta; Q; R \vdash_L \{ I \} S \{ I \} \\
\Delta; B; R \vdash_L \{ I \} \text{ loop } S \{ Q \} & \quad \text{(L: Loop)} \\
\Delta; B; R \vdash_L \{ e \} \sigma + P \{ Q \} & \quad \Delta; B; R \vdash_L \{ \text{if } e \} \sigma + P \{ Q \} \\
\Delta; B; R \vdash_L \{ \text{else } e \} \sigma + P \{ Q \} & \quad \text{(L: If)} \\
\Delta(\alpha) = \forall z \bar{v}. (P_f z \bar{v}, Q_f z v) & \quad P \geq P_f (\sigma(\bar{x})) + A \quad \forall v. (Q_f y v + A \geq \lambda \sigma. Q \sigma[r \mapsto v]) \\
\Delta; B; R \vdash_L \{ P \} r \leftarrow f(\bar{x}) \{ Q \} & \quad \text{(L: Call)} \\
\Delta \cup \Delta'; B; R \vdash_L \{ P \} S \{ Q \} & \quad P_f \geq 0 \\
\forall f P_f Q_f. \Delta'(f) = \forall z \bar{v}. (P_f z \bar{v}, Q_f z v) & \quad \forall y \bar{v}. (\Delta \cup \Delta'; \bot; Q_f y \vdash_L \{ P_f y \bar{v} \} S_f \{ \bot \}) \\
\Delta; B; R \vdash_L \{ P \} S \{ Q \} & \quad \text{(L: Extend)} \\
P \geq P' & \quad Q' \geq Q \quad B' \geq B \quad \forall v. (R' v \geq R v) \\
\Delta; B; R \vdash_L \{ P \} S \{ Q \} & \quad \text{(L: Widen)} \\
\Delta; B; R \vdash_L \{ P \} S \{ Q \} & \quad x \in Q_0^+ \\
\Delta; B + x; R + x \vdash_L \{ P + x \} S \{ Q + x \} & \quad \text{(L: Frame)}
\end{align*}
\]

Figure 13: Rules of the Quantitative Hoare Logic
Example Applications

- Linear stack bound for Fib function.
- Linear stack bound for factorial.
- Logarithmic stack bound for binary search.
- Logarithmic stack bound for Quicksort.
- Backend for automated stack bounds tool.
- A stack bound for full CertiKOS.
- A powerful semantic tool.
- A common language for tools and proofs.
Cost Semantics
We need a precise definition of the “resource cost” of a program.

\[ S : = x \leftarrow E \mid \text{skip} \mid S; S \]
| \text{while } E \text{ do } S |
| \text{if } E \text{ then } S \text{ else } S |
Define configurations as: \((S, K, H)\).

\[
K := \text{KLoop } E S K \mid \text{KSeq } S K \mid \text{KEmpty}
\]

Define rules like:

\[
(S_1; S_2, K, H) \rightarrow (S_1, \text{KSeq } S_2 K, H) \quad \text{(S:SEQ1)}
\]

\[
(\text{skip}, \text{KSeq } S K, H) \rightarrow (S, K, H) \quad \text{(S:SEQ2)}
\]
\[
\begin{align*}
\mathbb{E}_H &= n \\
(x \leftarrow E, K, H) &\rightarrow (\text{skip}, K, H[x \leftarrow n]) \quad (S: \text{SET}) \\
\mathbb{E}_H &= 0 \\
(\text{while } E \text{ do } S, K, H) &\rightarrow (\text{skip}, K, H) \quad (S: \text{WHILE1}) \\
\mathbb{E}_H &\neq 0 \\
(\text{while } E \text{ do } S, K, H) &\rightarrow (S, \text{KLoop } E S K, H) \quad (S: \text{WHILE2}) \\
(\text{skip}, \text{KLoop } E S K, H) &\rightarrow (\text{while } E \text{ do } S, K, H) \quad (S: \text{WHILE3})
\end{align*}
\]
(while $x < 1$ do $x \leftarrow x + 1, KE, \{x \leftarrow 0\}$)
→ $(x \leftarrow x + 1, KL \ (x < 1) \ (x \leftarrow x + 1) \ KE, \{x \leftarrow 0\})$
→ $(\text{skip}, KL \ (x < 1) \ (x \leftarrow x + 1) \ KE, \{x \leftarrow 1\})$
→ $(\text{while } x < 1 \text{ do } x \leftarrow x + 1, KE, \{x \leftarrow 1\})$
→ $(\text{skip}, KE, \{x \leftarrow 1\})$

$S:\text{While2, S:Set, S:While3, S:While1}$
Cost Semantics

Program configurations are now: \((S, K, H, c)\). This \(c\) is *resource counter*, it changes as the program executes.

\[
S := \ldots \mid \text{tick}(n)
\]

\[
(\text{tick}(n), K, H, c) \rightarrow (\text{skip}, K, H, c - n) \quad (S:\text{TICK})
\]

All the rules get a side condition \(c \geq 0\).
Stuck Configurations

A configuration is *stuck* if it cannot execute further. Different kinds of stuckness exist:

- Memory error.
- Divisions by 0.
- Resource crashes.

We recognize resource crashes as configurations $(S, K, H, c)$ where $c < 0$. 
Identifying crashes lets us define safety:

A configuration $C$ is *safe* for $n$ steps if any execution sequence of $n$ steps *or less* starting in $C$ does not end as a resource crash.

This predicate is formally defined inductively using the small-step semantics.
Why Indexing the Definition?

- It lets us talk about diverging executions. (Think about stack usage.)
- It makes the soundness proof of our logic possible. (In the while case.)
Resource Cost of a Program

\[ C_S = \inf \{ c \mid \forall n. \text{safe}_n(S, \text{KEmpty}, H_0, c) \} \]
Resource Cost of a Program

\[ C_S = \inf \{ c \mid \forall n. \text{safe}_n(S, \text{KEmpty}, H_0, c) \} \]

Problems with this notion:

- \( \inf \) is a promiss for trouble.
- Explicit mention of \( \text{KEmpty} \) and \( H_0 \).
- In short: non compositional!
Soundness of the Logic
Revisit Semantic Validity

Remember \( \{\Phi\}S\{\Phi'\} \equiv \Phi(\sigma) \geq C_S + \Phi'(\sigma'). \)
Let’s be more thorough.
Revisit Semantic Validity

Remember \( \{ \Phi \} S \{ \Phi' \} \equiv \Phi(\sigma) \geq C_S + \Phi'(\sigma') \).
Let’s be more thorough.

A continuation \( K \) is \( \text{safe}_n \) for a potential \( \Phi' \) if
\[ \forall H. \Phi'(H) \leq c \implies \text{safe}_n(\text{skip}, K, H, c). \]
We write \( \text{safe}K_n(\Phi', K) \).
Revisit Semantic Validity

Remember \( \{\Phi\}S\{\Phi'\} \equiv \Phi(\sigma) \geq C_S + \Phi'(\sigma') \).
Let’s be more thorough.

A continuation \( K \) is safe\(_n\) for a potential \( \Phi' \) if \( \forall H \ c. \ \Phi'(H) \leq c \implies \text{safe}_{n}(\text{skip}, K, H, c) \).
We write \( \text{safe}_{K_n}(\Phi', K) \).

A triple \( \{\Phi\}S\{\Phi'\} \) is valid for \( n \) steps if \( \forall K \ H \ c \ k \leq n. \text{safe}_{K_k}(\Phi', K) \land \Phi(H) \leq c \implies \text{safe}_{k}(S, K, H, c) \).
One Remark

We have $\forall \Phi' \; n. \; \text{safe}_{K_n}(\Phi', \text{KEmpty})$.

So, $\{\Phi\}S\{\Phi'\}$ valid for every $n$ implies:

$$\forall n. \; \text{safe}_n(S, \text{KEmpty}, H_0, \Phi(H_0)).$$

That is, $C_S \leq \Phi(H_0)$.

Our semantic validity of triples is connected to the intuitive resource cost of a program.
Example Proof: The Sequence

\[
\frac{\{\Phi\}S_1\{\Phi'\} \quad \{\Phi'\}S_2\{\Phi''\}}{\{\Phi\}S_1;S_2\{\Phi''\}} \quad (Q:\text{SEQ})
\]

If the two premisses are valid for \(n\) steps, the conclusion will be too.

Let \(K\) a continuation safe for \(k \leq n\) steps, let \(\Phi(H) \leq c\). We want \(\text{safe}_k(S_1;S_2,K,H,c)\).
Proving $safe_k(S_1; S_2, K, H, c)$

- By definition of $safe_k$, we must show $safe_{k-1}(S_1, K\text{Seq} S_2 K, H, c)$. 

- By the first premiss, we now have to show $safe_{k-1}(\Phi', K\text{Seq} S_2 K, H, c)$.

- By definition of $safe_{k-1}$, we must show $safe_{k-2}(S_2, K, H, c')$, for $\Phi'(H') \leq c'$.

- The second premiss finishes the proof, since $safe_{k-2}(\Phi'', K)$ (after weakening).
Proving $safe_k(S_1; S_2, K, H, c)$

- By definition of $safe_k$, we must show $safe_{k-1}(S_1, KSeq S_2 K, H, c)$.

- By the first premiss, we now have to show $safeK_{k-1}(\Phi', KSeq S_2 K)$. 
Proving $safe_k(S_1; S_2, K, H, c)$

- By definition of $safe_k$, we must show $safe_{k-1}(S_1, K\text{Seq } S_2 K, H, c)$.

- By the first premiss, we now have to show $safe_{K_{k-1}}(\Phi', K\text{Seq } S_2 K)$.

- By definition of $safe_{K_{k-1}}$, we must show $safe_{k-1}(\text{skip, KSeq } S_2 K, H', c')$ for $\Phi'(H') \leq c'$. 
Proving $safe_k(S_1; S_2, K, H, c)$

- By definition of $safe_k$, we must show $safe_{k-1}(S_1, K\text{Seq} S_2 K, H, c)$.

- By the first premiss, we now have to show $safe_{K_{k-1}}(\Phi', K\text{Seq} S_2 K)$.

- By definition of $safe_{K_{k-1}}$, we must show $safe_{k-1}(\text{skip}, K\text{Seq} S_2 K, H', c')$ for $\Phi'(H') \leq c'$.

- By definition of $safe_{k-1}$, we must show $safe_{k-2}(S_2, K, H, c')$, for $\Phi'(H') \leq c'$.
Proving $safe_k(S_1; S_2, K, H, c)$

- By definition of $safe_k$, we must show $safe_{k-1}(S_1, \text{KSeq } S_2 K, H, c)$.
- By the first premiss, we now have to show $safe_{K_{k-1}}(\Phi', \text{KSeq } S_2 K)$.
- By definition of $safe_{K_{k-1}}$, we must show $safe_{k-1}(\text{skip, KSeq } S_2 K, H', c')$ for $\Phi'(H') \leq c'$.
- By definition of $safe_{k-1}$, we must show $safe_{k-2}(S_2, K, H, c')$, for $\Phi'(H') \leq c'$.
- The second premiss finishes the proof, since $safe_{K_{k-2}}(\Phi'', K)$ (after weakening).
This Seemed Tricky?

It’s not! The Coq proof we have for the logic might be the smallest soundness proof for a program logic for C in Coq! (400 lines)

Very much recommended for novices and teaching!
Proof with try (intros; ksgn).
unfold valid at 3, safe; intros.
assert (CNNEG: 0 ≤ c).
{ eapply (valid_nneg n B R P Q’ s1 x XSGN PRE1)...
eapply (valid_nneg n B R Q’ Q s2 x XSGN PRE2)...
eassumption. }
split; [ exact CNNEG | step ].
apply PRE1 with (x := x); try (omega || assumption).
clearINI.
unfold safek, safe; intuition;
try step; try ksgn.
+ eapply SAFEK; assumption.
+ simpl; eapply SAFEK; auto.
+ eapply (valid_nneg n B R Q’ Q s2 x XSGN PRE2)...
eassumption.
+ eapply PRE2 with (x := x); try (omega || apply INI).
simpl; eapply SAFEK; now auto.
+ eapply (valid_nneg n B R Q’ Q s2 x XSGN PRE2)...
eassumption.
Qed.

The previous proof in the full Clight context in Coq. Arguably very very short!
Automatic Derivation of Resource Bounds
A potential function can be any function. What if we only look at a few of them?
Automating the Proof Search

A potential function can be any function. What if we only look at a few of them?

$$\Phi(H) = k_0 + \sum_{x,y} k_{xy} \cdot |[H(x), H(y)]|$$

where $$k_\_ \in \mathbb{Q}_0^+$$ and $$|[a, b]| = \max(b - a, 0)$$.
Motivation for Intervals

\[ \Phi(H) = \frac{1}{2} |[0, H(y)]| \]

```c
for (x = 0; x < y - 1; x += 2) {
    tick(1);
}
```
Motivation for Intervals

\[
\Phi(H) = \frac{1}{2} |[0, H(y)]|
\]

for \((x = 0; x < y-1; x += 2)\) {
\text{\texttt{tick(1);}}
}


Indices

To succintly refer to a potential function, we use indices.

\[ \mathcal{I} = \{0\} \cup \{xy \mid x, y \in \text{Vars}\} \]
To succinctly refer to a potential function, we use indices.

\[ \mathcal{I} = \{0\} \cup \{xy \mid x, y \in \text{Vars}\} \]

If \( f_0 = \lambda_1 \) and \( f_{xy} = \lambda H. |[H(x), H(y)]| \)

\[ \Phi(H) = \sum_{I \in \mathcal{I}} k_I \cdot f_I(H). \]

\( f_I \) is a base function. (Linear algebra.)
Rules for Potential

Idea: Reuse the logic’s rules for soundness and constrain potential functions on their coefficients.

- For example: Ensure $\Phi > \Phi'$ by $\forall I, k_I > k'_I$.
- Reuse all syntax directed rules as-is.
- Add a little more work for statements modifying the heap.
Increments of Variables

Consider the increment program \( x \leftarrow x + 1 \).

The logic rule is notoriously unhelpful:

\[
\{ \lambda H. \Phi( H[x \leftarrow \llbracket E \rrbracket_H] ) \} x \leftarrow E\{ \Phi \} \quad (Q:\text{SET})
\]

We need to understand how \( \Phi = \sum_I k_I \cdot f_I \) is changed.
Constraints for \( \{ \Phi \} x \leftarrow x + 1 \{ \Phi' \} \)

Only \([y, x]\) and \([x, y]\) will change. We write \(x'\) for the new value of \(x\).
Constraints for \( \{ \Phi \} x \leftarrow x + 1 \{ \Phi' \} \)

Only \([y, x]\) and \([x, y]\) will change. We write \(x'\) for the new value of \(x\).

- Consider \(\Phi = k_0 + k_{yx} \cdot |[y, x]|\), we have \(|[y, x']| = |[y, x]| + 1\), so \(\Phi' = (k_0 - k_{yx}) + k_{yx} \cdot |[y, x]|\).
Constraints for $\{\Phi\}x \leftarrow x + 1\{\Phi'\}$

Only $[y, x]$ and $[x, y]$ will change. We write $x'$ for the new value of $x$.

- Consider $\Phi = k_0 + k_{yx} \cdot |[y, x]|$, we have $|[y, x']| = |[y, x]| + 1$, so $\Phi' = (k_0 - k_{yx}) + k_{yx} \cdot |[y, x]|$.

- Suppose $x' \in [x, y]$ and consider $\Phi = k_0 + k_{xy} \cdot |[x, y]|$, we have $|[x', y]| = |[x, y]| - 1$, so $\Phi' = (k_0 + k_{xy}) + k_{xy} \cdot |[x, y]|$. 
Constraints for $\{\Phi\}x \leftarrow x + 1\{\Phi'\}$

Only $[y, x]$ and $[x, y]$ will change. We write $x'$ for the new value of $x$.

- Consider $\Phi = k_0 + k_{yx} \cdot ||y, x||$, we have $||y, x'|| = ||y, x|| + 1$, so $\Phi' = (k_0 - k_{yx}) + k_{yx} \cdot ||y, x||$.

- Suppose $x' \in [x, y]$ and consider $\Phi = k_0 + k_{xy} \cdot ||x, y||$, we have $||x', y|| = ||x, y|| - 1$, so $\Phi' = (k_0 + k_{xy}) + k_{xy} \cdot ||x, y||$.

In both cases $\Phi(H) = \Phi'(H[x \leftarrow x + 1])$. 
Figure 4: Inference rules of the quantitative analysis.
Example Derivation

\[
\{\cdot; 0 + T \cdot |[x, y]|\}
\]

while \(x < y\) \{
\{x < y; 0 + T \cdot |[x, y]|\}
\}
\[
x = x + 1 ;
\{x \leq y; T + T \cdot |[x, y]|\}
\]
\[
tick(T);
\{x \leq y; 0 + T \cdot |[x, y]|\}
\}
\[
x \geq y; 0 + T \cdot |[x, y]|\}
\]
Tarjan’s Example From 50 Slides Ago

\[
\{\cdot; 2 \cdot \|[0,n]\| + \|[0,s]\|\}
\]

while (n > 0) {
\[
\{n > 0; 2 \cdot \|[0,n]\| + \|[0,s]\|\}
\]

n--; \\
\[
\{\cdot; 2 + 2 \cdot \|[0,n]\| + \|[0,s]\|\}
\]

if (*)
\[
\text{tick (1); } \{\cdot; 2 \cdot \|[0,n]\| + \|[0,s]\|\}
\]

else
\[
\{\cdot; 0 + 2 \cdot \|[0,n]\| + \|[0,s]\|\}
\]

while (s > 0)
\[
\{s > 0; 2 \cdot \|[0,n]\| + \|[0,s]\|\}
\]

s--; \\
\[
\{\cdot; 1 + 2 \cdot \|[0,n]\| + \|[0,s]\|\}
\]

\text{tick (1); } \{\cdot; 2 \cdot \|[0,n]\| + \|[0,s]\|\}
}
How to Automate?

Remark: All the constraints generated are linear.

- Apply the rules with dummy names for \((k_I)_I\).
- Collect all the constraints.
- Feed them the an LP solver.
- A solution is a proof certificate.
- No solution, report an error.

Implemented in \(C^4B\), validated by the PLDI AEC 2015.
Future Work and Demo
What Now?

- (P) Automation for polynomial bounds.
- (TP) Extend automation to handle memory.
- (T) Prove logic completeness.
- (T) Integrate with contextual refinement.
- (P) Apply to real-time systems.

T is for theory, P is for practice.