On the Length of Instruction Sequences for C

Sjoerd Bouber

June 17, 2015
Abstract

In [3] an algebra of finite instruction sequences is introduced by presenting the C semigroup, a mathematical representation for imperative sequential programs. C-programs can be represented without directional bias. C has both forward and backward instructions and a C-expression can be interpreted starting at any instruction. C is an alternative to ProGram Algebra[2, 4, 1] (PGA). Both C and PGA are tools that aid in the research of the fundamental properties of imperative sequential programming. Unlike C, PGA uses infinite instruction sequences to model infinite behaviour. In this sense C seems to be a more realistic approach to model finite imperative sequential programs. To formally describe the semantics of instruction sequences Basic Thread Algebra (BTA) is used. This thesis explores an approach to minimize the amount of C-instructions needed to code a regular thread using a uniform bound that depends on the number of states.
## Contents

1 Introduction 5

2 Basic Thread Algebra 7
   2.1 Finite Threads 7
   2.2 Infinite Threads 8

3 The Code Semigroup $C$ 9
   3.1 Instruction Sequences 9
   3.2 The Semigroup $C$ 9
   3.3 Thread Extraction for $C$-inseqs 10

4 Bounds on $C$-Program Length for Threads 13
   4.1 State Chaining 13
   4.2 Threads as Directed Graphs 14
   4.3 Spanning Trees 15
   4.4 An Upper Bound Using Neighbour Pairs 16

5 Discussion 19
   5.1 PGA 19
   5.2 Further Work 19

Bibliography
In *An instruction Sequence Semigroup with Involutive Anti-Automorphisms* [3] Bergstra and Ponse introduce an algebra of finite instruction sequences by presenting a semigroup $C$, a mathematical representation for imperative sequential programs. C-programs can be represented without directional bias. $C$ has both forward and backward instructions and a $C$-expression can be interpreted starting at any instruction. $C$ is an alternative to ProGram Algebra [2, 4, 1] (PGA). Both $C$ and PGA are tools that aid in the research of the fundamental properties of imperative sequential programming. Unlike $C$, PGA uses infinite instruction sequences to model infinite behaviour. In this sense $C$ seems to be a more realistic approach to model finite imperative sequential programs. To formally describe the semantics of instruction sequences Basic Thread Algebra ($BTA$) is used. Chapter 2 describes $BTA$ and defines the concept of a thread. In Chapter 3 the $C$ semigroup is defined and equations are defined to describe the semantics using $BTA$. The final chapter explores an approach to minimize the amount of $C$-instructions needed to code a regular thread using a uniform bound that depends on the number of states.
CHAPTER 2

Basic Thread Algebra

2.1 Finite Threads

Basic Thread Algebra (BTA) is a form of process algebra that is suitable to describe the behaviour of sequential programs. It is assumed we have a set of actions $A$, often kept implicit. Actions are executed by some execution environment and yield a boolean value true or false upon execution. This reply determines how the execution should proceed. BTA expressions called threads are built using the constants $S$, $D$ and the postconditional composition operator:

- The termination constant $S \in BTA$
- The deadlock constant $D \in BTA$
- The postconditional composition operator $\preceq \preceq : BTA \times A \times BTA \rightarrow BTA$

We will often write $P$ or $Q$ for an arbitrary thread and $a$ for an arbitrary action. The postconditional composition operator $P \preceq a \succeq Q$ prescribes execution of action $a$, and then to continue execution with thread $P$ if the reply to action $a$ is true, otherwise execution proceeds with thread $Q$. We define action prefixing $a \circ P$ as an abbreviation for $P \preceq a \succeq P$. Action prefixing binds stronger than postconditional composition.

![Graphical representation of the thread (D $\preceq b \succeq (c\circ S)) \preceq a \succeq D$. An action between angular brackets represents the postconditional operator and an action between square brackets represents the prefixing operator.](image)

Upon execution each BTA thread performs a finite amount of actions before termination or deadlock follows.
2.2 Infinite Threads

In order to define infinite threads we will first define the approximation operator $\pi : \mathbb{N} \times BTA \rightarrow BTA$ which gives the behaviour of a thread up to a specific depth as follows:

1. $\pi(0, P) = D$
2. $\pi(n + 1, S) = S$
3. $\pi(n + 1, D) = D$
4. $\pi(n + 1, P \leq a \geq Q) = \pi(n, P) \leq a \geq \pi(n, Q)$

for $P, Q \in BTA$ and $n \in \mathbb{N}$.

We define $BTA^\infty$ which also includes infinite threads as the complete partial order of projective sequences of finite threads (see [1]):

$$BTA^\infty = \{(P_n)_{n \in \mathbb{N}} \mid \forall n \in \mathbb{N}(P_n \in BTA \& \pi(n, P_{n+1}) = P_n)\}$$

**Definition 2.2.1.** The set $Res(P)$ of residual threads of $P \in BTA^\infty$ is inductively defined as:

1. $P \in Res(P)$
2. $Q \leq a \geq R \in Res(P)$ implies $Q \in Res(P)$ and $R \in Res(P)$

A thread $P$ is regular if $Res(P)$ is finite.

Each regular thread $P \in BTA^\infty$ can be described as a finite set of equations (see [1]).

**Definition 2.2.2.** A finite linear recursive specification $E$ of length $n$ over $BTA^\infty$ with indices $I = \{1 \ldots n\}$ is a set of equations $\{P_i = t_i \mid i \in I\}$ with thread identifiers or states $P_i$ and each term $t_i$ of form $S, D$ or $P_j \leq a \geq P_k$ with $j, k \in I$ and $a \in A$.

A thread identifier $P_i$ is directly reachable from $P_j$ if $P_j = P_i \leq a \geq P_k$ or $P_j = P_k \leq a \geq P_i$ for some $k \in I$.

The finite linear recursive specification (2.1) provides an example of a regular thread consisting of the states $P_1, P_2, P_3$ and $P_4$:

$$P_1 = a \circ P_2$$
$$P_2 = P_1 \leq b \geq P_3$$
$$P_3 = P_4 \leq c \geq P_2$$
$$P_4 = P_3 \leq d \geq P_1$$

\begin{equation}
(2.1)
\end{equation}

Figure 2.2: Graphical representation of the thread described in the finite linear recursive specification of (2.1).
CHAPTER 3

The Code Semigroup C

3.1 Instruction Sequences

In this chapter we will first introduce the notion of instruction sequences. Subsequently we introduce a 
semigroup of specific instruction sequences, the C semigroup, which is central to this thesis.

Let \( I \) be a non-empty set of instructions and \( ; \) the concatenation operator, an associative binary operator 
on \( I \). We will first inductively define an instruction sequence (inseq), as the concatenation of instructions:

\[
I_1 = I \\
I_{n+1} = \{ X; u \mid X \in I^n, u \in I^1 \}
\]

An inseq \( X \) of length \( n \) is an element of \( I^n \), of which the length is denoted as \( \ell(X) = n \).

The code semigroup \( I^+ \) is generated by the set of instructions i.e. it consists of all possible concatenations 
of a finite number of instructions in \( I \). Formally, \( I^+ = (I, ;) \).

3.2 The Semigroup C

For the semigroup C we have two types of instructions, forward and backward instructions. An implicit 
parameter of C is the set of actions \( A \). We assume actions are executed by an execution environment 
and this environment returns either true or false after executing an action.

The set of C-instructions \( I_C \) contains the following elements for \( k \in \mathbb{N}^+ \) and \( a \in A \):

- /\( a \) is the forward basic instruction, it specifies to perform action \( a \) and then if there is an instruction 
on the right-hand side it specifies to continues execution with that instruction. If there is no such in-
struction deadlock is prescribed.

- +/\( a \) is the forward positive test instruction, it specifies to perform action \( a \). If the execution envi-
ronment returns true the instruction concatenated to its right-hand side is executed next and otherwise 
the second instruction concatenated to its right is executed next. Deadlock follows if there is no such 
instruction.

- −/\( a \) is the forward negative test instruction, in the false case the next instruction is the instruction 
catenated to its right-hand side and in the true case it’s the second instruction concatenated to its 
right-hand side. Deadlock follows if there is no such instruction.
slash / is the forward jump instruction, it specifies to execute the instruction k positions to the right or prescribes deadlock if no such instruction exists.

\( a, +a, \neg a, \neg \# k \) are the backward counterparts and mirror the behaviour of the corresponding forward instructions.

! is the termination instruction and prescribes successful termination.

\# is the abort instruction and specifies deadlock.

In the case where there is a cycle of jump instructions where no actions are executed deadlock follows.

**Examples**

The C-inseq /a; /# 2; /b; /c; \# 2; ! prescribes to perform an action \( a \) and proceeds with jumping to a forward positive test of action \( c \). If this action yields true a jump to the forward basic instruction \( b \) is executed, on returning false termination follows.

The C-inseq ++/a; /# 2; /b; \# 2 prescribes to perform a forward positive test of action \( a \). If the test yields true deadlock follows because there is a loop of jump instructions without any actions. When the test returns false the \( b \) action is executed first followed by deadlock.

The C-inseq /a; /b; /# 2; /c prescribes to perform actions \( a \) and \( b \) followed by jumping two instructions to the right. Since there is no such instruction deadlock follows.

### 3.3 Thread Extraction for C-inseqs

The semantics described above can be formally specified by specifying a thread extraction operation \( |X|_C \) for a C-inseq \( X \) and defining \( |X|_C = |X|_C^1 \) where the auxiliary operator \( |X|_C^i \) with \( i \) an integer value is defined as follows:

\[
|X|_C^i = \begin{cases} 
D & \text{if } i < 1 \text{ or } i > \ell(X) \\
a \circ |X|_C^{i+1} & \text{if } X_i = /a \\
a \circ |X|_C^{-1} & \text{if } X_i = \neg a \\
|X|_C^{i+1} \leq a \geq |X|_C^{i+2} & \text{if } X_i = +/a \\
|X|_C^{i-1} \leq a \geq |X|_C^{i+2} & \text{if } X_i = +\neg a \\
|X|_C^{i+2} \leq a \geq |X|_C^{i+1} & \text{if } X_i = -/a \\
|X|_C^{i-2} \leq a \geq |X|_C^{i-1} & \text{if } X_i = -\neg a \\
|X|_C^{i+k} & \text{if } X_i = /\# k \\
|X|_C^{-k} & \text{if } X_i = \neg \# k \\
D & \text{if } X_i = # \\
S & \text{if } X_i = ! 
\end{cases}
\]
Examples

\[ |a; \#2; /b; +/c; \#2; |c = P \]

where \( P \) is defined by:

\[
\begin{align*}
P &= a \circ Q \\
Q &= R \sqsubseteq c \sqsupseteq T \\
R &= b \circ Q \\
T &= S
\end{align*}
\]

\[ |+ /a; /\#2; /b; /\#2|c \]

where \( P \) is defined by:

\[
\begin{align*}
P &= Q \sqsubseteq a \sqsupseteq R \\
Q &= D \\
R &= b \circ Q
\end{align*}
\]

\[ |/a; /b; /\#2; /c|c = P \]

where \( P \) is defined by:

\[
\begin{align*}
P &= a \circ Q \\
Q &= b \circ R \\
R &= D
\end{align*}
\]
CHAPTER 4

Bounds on C-Program Length for Threads

Given a regular thread of \( n \) states, we can always find more than one C-inseq with a matching behavior. In [3] the problem of defining bounds on the number of C-instructions needed to code such a thread is stated. The observation that we can code each state as either \(#, \!\) or a positive test instruction \(+/a\) concatenated with jumps to successive states results in an upper bound of \( 3n \) instructions was made in [3].

Formally, for a thread \( P \) with states \( P_1 \ldots P_n \) we construct a C-inseq \( X = X_1; \ldots; X_n \) as follows:

\[
X_i = \begin{cases} 
!; \#; \# & \text{if } P_i = S \\
\#; \#; \# & \text{if } P_i = D \\
+/a; J(X_j); J(X_k) & \text{if } P_i = P_j \sqsubseteq a \sqsupseteq P_k
\end{cases}
\]

where \( J(X_i) \) is a forward or backward jump to the start position of \( X_i \).

It is clear that \( \ell(X) = 3n \) and \( |X|_C = P_i \) so \( |X|_C = P \).

4.1 State Chaining

Above we showed that each state can be coded as 3 C-instructions. Observe that the second jump of \( X_i \) can be coded as \(/\#1\) if \( P_{i+1} \) is directly reachable from \( P_i \), replacing \(+/a\) with \(-/a\) if needed. This results in an instruction we can omit, reducing the total number of instructions by one for each two states where this is the case. Now reducing the maximum number of instructions we need to code an arbitrary thread comes down to reordering the states.

For example, consider the following regular thread:

\[
P_1 = P_2 \sqsubseteq a \sqsupseteq P_3 \\
P_2 = b \circ P_4 \\
P_3 = P_1 \sqsubseteq c \sqsupseteq P_2 \\
P_4 = S
\]

The resulting C-inseq \( X \) consists of 12 instructions:

\[
X = +/a; /\#2; /\#4; +/b; /\#5; /\#4; +/c; \#7; \#5; !; \#; \\
\]

Observe \( P_2 \) is directly reachable from \( P_1 \) and \( P_4 \) is directly reachable from \( P_2 \). Using state chaining we can reorder the instructions as follows without altering the behavior:

\[
|X|_C = | - /a; /\#8; /\#1; +/b; /\#2; /\#1; !; \#; +/c; \#10; \#8|_C \\
= | - /a; /\#6; +/b; /\#1; !; \#; +/c; \#8; \#7|_C
\]
In this particular example the deadlock instructions following the termination instruction could also be omitted (which then requires to adjust the jump counters).

After reordering the resulting C-inseq consists of 10 C-instructions. More generally, if we have a sequence of \( n \) distinct states \( P_1 \ldots P_n \) such that \( P_i \) is directly reachable from \( P_{i-1} \) for \( 0 < i \leq n \) then we can code those states in \( 2n + 1 \) C-instructions. A straightforward consequence is that we can reorder the states of any thread with \( n \geq 2 \) states in such a way that we can omit one jump instruction, resulting in an upper bound of \( 3n - 1 \) C-instructions. In the following sections this basic idea of reordering states in such a way that we can omit as many jump instructions as possible is further explored.

4.2 Threads as Directed Graphs

To simplify the idea of state chaining we will leave out the actions and view a thread as a directed graph. In such a graph each state is represented by a vertex. Two vertices are connected by an arc if the states they represent are directly reachable.

**Definition 4.2.1.** A directed graph is a pair \( G = (V, A) \) with \( V \) a set of vertices or nodes and \( A \) a set of ordered pairs or arcs \((v_i, v_j)\) where \( v_i, v_j \in V \).

**Definition 4.2.2.** A path of length \( k \) is an alternating sequence of distinct vertices and arcs \( v_0, a_0, v_1, a_1, v_2 \ldots, v_k \) where \( v_i \in V \) and \( a_i \in A \).

**Definition 4.2.3.** For a directed graph \( G = (V, A) \) with vertices \( v, w \in V \), we say \( w \) is reachable from \( v \) if there exists a path from \( v \) to \( w \). We also write \( v \rightarrow w \) if \( w \) is reachable from \( v \). Furthermore we write \( v \leftrightarrow w \) if \( w \) is directly reachable from \( v \), that is, \((v, w) \in A \).

**Definition 4.2.4.** The indegree \( \deg^-(v) \) of a vertex \( v \) is defined as \(|\{w \mid (w, v) \in A\}|\) and similarly the outdegree \( \deg^+(v) \) is defined as \(|\{w \mid (v, w) \in A\}|\). A vertex with \( \deg^-(v) = 0 \) is called a source and a vertex with \( \deg^+(v) = 0 \) is called a sink.

Note that the directed graphs which represent threads have a maximum outdegree of 2 and the vertices which act as the deadlock and termination states are sinks. For the threads we model we assume there exists one state from which every other state can be reached. If such a state does not exist, we can simply split the thread into several threads for which this assumption holds.

**Definition 4.2.5.** A directed graph \( G = (V, A) \) is said to be uni-reachable if there is a \( v \in V \) such that for every \( w \in V \) if \( w \neq v \) then there exists a path from \( v \) to \( w \).

For example, the following thread \( P \) produces the uni-reachable directed graph below in Fig 4.1:

\[
\begin{align*}
P &= Q \leq a \geq R \\
Q &= D \\
R &= b \circ T \\
T &= P \leq c \geq U \\
U &= U \leq d \geq Q
\end{align*}
\]
4.3 Spanning Trees

In this section we will further simplify the directed graphs by removing all cycles and loops. The resulting graph is a directed acyclic graph (DAG).

**Definition 4.3.1.** A directed acyclic graph (DAG) is a directed graph that contains no cycles.

The final step of simplification is to remove edges until each vertex has a maximum indegree of 1. This results in a spanning tree which is a binary tree since the maximum outdegree is equal to 2. For some examples see Fig. 4.3.
Figure 4.3. Two spanning trees of the graph in Fig. 4.2 with $P$ as root.

The resulting spanning tree of a graph representing a thread containing $n$ states has a depth of $d \geq \log_2(n)$. This gives us a path of length $d$, thus we can code any thread of $n$ states in $3n - \lceil \log_2(n) \rceil + 1$ C-instructions. In the next section we narrow down the amount of C-instructions by finding a number of distinct pairs of directly connected vertices.

4.4 An Upper Bound Using Neighbour Pairs

**Definition 4.4.1.** $N \subset A$ is a set of distinct neighbour pairs of a directed graph $G = (V, A)$ if for all $n = (v_i, v_j) \in N$ it holds that $(v_i, v_k) \in N \implies v_j = v_k$ and $(v_k, v_j) \in N \implies v_i = v_k$.

We will prove that given a spanning tree, we can find $\lfloor (n + 2)/3 \rfloor$ distinct neighbour pairs which implies we can code any thread of $n$ states in $3n - \lfloor (n + 2)/3 \rfloor$ C-instructions.

**Theorem 4.4.1.** Every binary tree $T = (V, A)$ of $n > 2$ nodes with root $r$ contains $\lfloor (n + 2)/3 \rfloor$ distinct neighbour pairs.

**Proof.** We will prove this using induction on $n$. The base case $n = 3$ trivially contains 1 neighbour pair, as required. Now suppose the theorem holds for all values $n$ up to some $k$, $k \geq 3$.

Inductive step: let $n = k + 1$. Since $n > 3$ there must be a leaf $v \in V$ such that $w \mapsto v$ with $w \in V$ and $w \neq r$ and that satisfies one of the following two cases:

Case 1: there is a leaf $t \in V$ such that $w \mapsto t$ and furthermore $t \neq v$. Then we have a neighbour pair $(v, w)$ and by IH we find for the reduced tree that does not contain the nodes $w, v, t$, $\lfloor k/3 \rfloor$ dp’s. Hence, $T$ contains $1 + \lfloor k/3 \rfloor = \lfloor (n + 2)/3 \rfloor$ dp’s.

Case 2: for no node $t \neq v, w \mapsto t$. Then we have a neighbour pair $(v, w)$ and by IH we find for the reduced tree that does not contain the nodes $w, v, [(k+1)/3]$ dp’s. Hence, $T$ contains $1 + [(k+1)/3] \geq \lfloor (n+2)/3 \rfloor$ dp’s.
Corollary 4.4.1 Every uni-reachable regular thread $P$ of $n$ states can be coded in $3n - \lfloor (n + 2)/3 \rfloor$ C-instructions.

Proof. Determine a spanning tree $T$ of $P$ and a set $N$ of dnp’s in $T$. As pointed out in Section 4.1, coding a neighbour pair requires 5 C-instructions, and each state that is not in $N$ can be coded with 3 C-instructions. Hence, we find that $P$ can be coded in

$$\lfloor (n + 2)/3 \rfloor \cdot 5 + (n - 2 \cdot \lfloor (n + 2)/3 \rfloor) \cdot 3 = 3n - \lfloor (n + 2)/3 \rfloor$$

C-instructions. \qed
CHAPTER 5

Discussion

In this thesis we described the C semigroup and the procedure to extract the behavior of C-inseqs using Basic Thread Algebra. We have explored a model to represent threads as directed graphs. To improve the upper bound on the amount of instructions needed to code a regular thread we introduced the notion of distinct neighbour pairs. Using distinct neighbour pairs we proved that any regular thread of \( n \) states can be coded in \( 3n - \lfloor (n + 2)/3 \rfloor \) C-instructions.

5.1 PGA

As described in the introduction, PGA is an earlier approach to model imperative sequential programs. C’s syntax is similar to PGA, except for the fact that all C-instructions that prescribe further control include a particular direction of control (left or right). PGA contains forward basic, test and jump instructions defined in the same way as C. In [2] several program notations are specified which can be translated into PGA. The semantics of one of those languages, PGLB, is almost identical to C. By defining a homomorphism \( \phi : C' \to PGLB \), where \( \mathcal{I}_{C'} = \mathcal{I}_C \setminus \{/a, \backslash a, +/a, -/a\} \), all results on the C-inseq lengths for regular threads also hold for PGLB. \( \phi \) is defined as follows:

1. \( \phi(+/a) = +a \)
2. \( \phi(-/a) = -a \)
3. \( \phi(\backslash #) = \backslash # \)
4. \( \phi(/#) = # \)
5. \( \phi(#) = #0 \)
6. \( \phi(!) = ! \)
7. \( \phi(x; y) = \phi(x); \phi(y) \)

5.2 Further Work

• Similarly to neighbour pairs, neighbour triples or a combination of both could be used to further improve the upper bound on the amount of C-instructions needed to code particular threads. As described in Section 4.1, each such triple of states can be coded in 7 C-instructions. An alternative where not a single path but multiple paths are found in the spanning tree could potentially improve the upper bound even more.

• The coding of states as described in Chapter 4 does not use the backward test instructions +\( \backslash a \) and -\( \backslash a \) or the basic instructions /\( a \) and \( \backslash a \). Using those C-instructions could potentially reduce the amount of C-instructions needed to code states.
Bibliography


