Expressiveness and Extensions of an Instruction Sequence Semigroup

MSc Thesis (*Afstudeerscriptie*)

written by

S.H.P. Schroevers
(born February 20, 1985 in Zaandam, The Netherlands)

under the supervision of **Dr. Alban Ponse**, and submitted to the Board of Examiners in partial fulfillment of the requirements for the degree of

MSc in Logic

at the *Universiteit van Amsterdam*.

Date of the public defense:
February 15, 2010

Members of the Thesis Committee:
Dr. Inge Bethke
Prof. Dr. Peter van Emde Boas
Dr. Alban Ponse
Prof. Dr. Frank Veltman

Institute for Logic, Language and Computation

Abstract

PGA, short for ProGram Algebra [PvdZ06, BL02], describes sequential programs as finite or infinite (repeating) sequences of instructions. The semigroup C of finite instruction sequences [BP09a] was introduced as an equally expressive alternative to PGA. PGA instructions are executed from left to right; most C instructions come in a left-to-right as well as a right-to-left flavor. This thesis builds on C by introducing an alternative semigroup Cg which employs label and goto instructions instead of relative jump instructions as control structures. Cg can be translated to C and vice versa (and is thus equally expressive). It is shown that restricting the instruction sets of C and Cg to contain only finitely many distinct jump, goto or label instructions in either or both directions reduces their expressiveness. Instruction sets with an infinite number of these instructions in both directions (not necessarily all such instructions) do not suffer a loss of expressiveness.

Contents

Chapter 1

Introduction

Bergstra and Ponse [BP09a] introduce an algebra of finite instruction sequences by presenting a semigroup C in which programs can be represented without directional bias: in terms of the next instruction to be executed, C has both forward and backward instructions and a C-expression can be interpreted starting from any instruction.

[BP09a] provides equations for thread extraction, i.e. C's program semantics, and defines behavioral equivalence. It considers thread extraction compatible (anti-)homomorphisms and (anti-)automorphisms. Lastly, it discusses some expressiveness results.

C is a recent alternative to PGA [PvdZ06, BL02], short for ProGram Algebra. Contrary to C, PGA uses infinite instruction sequences to model infinite behavior. Since both PGA and C are tools that aid in the research on imperative sequential programming, and given that any "real world" programs are always finite, C appears to be a more realistic approach to a mathematical representation for sequential programs.

This thesis introduces PGA and C and describes their semantics. It then defines an alternative to C called Cg which uses label and goto instructions as control structures, as opposed to C's relative jump instructions. Behavior preserving mappings are defined between PGA, C and Cg, thereby establishing that they are equally expressive.

The final chapter of this thesis investigates the expressiveness of subsemigroups of C and Cg, particularly those from which a finite or infinite number of jump or goto instructions has been removed, thereby improving on an expressiveness result presented in [BP09a].

Lastly, the reader should take note of Appendix A, which provides a graphical representation of some of the (single-pass) instruction sequences defined in this thesis and the mappings between them.

Chapter 2

Preliminaries

In this chapter we introduce the concepts on which the remainder of this thesis builds. In §2.1 basic thread algebra is introduced. This allows us to describe the semantics of instruction sequences. Next, §2.2 and §2.3 introduce two different takes on the way in which instruction sequences can be represented: on the one hand there is PGA which describes finite or infinite single-pass instruction sequences; on the other hand we can take the (arguably more natural) stance that all instruction sequences must be finite while allowing instructions to be executed multiple times. It is the latter theory which describes instruction sequence semigroups, two concrete instances of which will be introduced in the following chapters as C and Cg.

2.1 Basic Thread Algebra

Basic thread algebra, BTA for short, is a means to describe the behavior of sequential programs upon execution. BTA takes the position that program execution consists of a sequence of *basic actions* which are performed inside some execution environment. It is assumed that a fixed but arbitrary set of basic actions $\mathcal{A}$ is specified; this parameter is often kept implicit. Upon execution of an action the execution environment yields a boolean reply, the value of which specifies how execution should proceed.

In this section we will briefly introduce basic thread algebra. For more on this subject we refer to [PvdZ06, BP09a, BL02][1].

BTA expressions are called *threads*. The set of all threads is denoted BTA. For any set $\mathcal{A}$, threads are built using two constants and a single ternary operator:

- The *deadlock* constant D: BTA.
- The *termination* constant S: BTA.
- The *postconditional composition* operator $\_ \trianglelefteq \_ \trianglerighteq \_$: BTA $\times\, \mathcal{A} \times$ BTA $\to$ BTA.

It follows that each closed BTA expression performs finitely many actions and then terminates or becomes inactive (in the case of deadlock).

For $P \in$ BTA and $a \in \mathcal{A}$, the thread $P \trianglelefteq a \trianglerighteq P$ is often more conveniently denoted $a \circ P$. The *action prefix* operator $\circ$ can be used only if the boolean reply returned after execution of a does not influence further behavior. Action prefix binds stronger than postconditional composition. Additionally, for all $n \geq 1$ we will define $a^n \circ P$ to mean the thread which performs n a-actions, followed by the behavior described by the thread P. That is, $a^1 \circ P = a \circ P$ and $a^{n+1} \circ P = a \circ (a^n \circ P)$.

[1] In [BL02] BTA is called BPPA.

The *approximation operator* $\pi\colon \mathbb{N} \times \mathrm{BTA} \to \mathrm{BTA}$ returns the behavior of a given thread up to a specified "depth"[2], i.e., it bounds the number of actions performed. For all $P, Q \in \mathrm{BTA}$ and $a \in \mathcal{A}$ we define,

$$\begin{aligned} \pi(0, P) &= \mathsf{D} \\ \pi(n+1, \mathsf{S}) &= \mathsf{S} \\ \pi(n+1, \mathsf{D}) &= \mathsf{D} \\ \pi(n+1, P \trianglelefteq a \trianglerighteq Q) &= \pi(n, P) \trianglelefteq a \trianglerighteq \pi(n, Q) \end{aligned}$$

From now on we will write $\pi_n(P)$ instead of $\pi(n, P)$ for brevity. Since every BTA thread is finite, it follows that for every $P \in \mathrm{BTA}$ there exists some $n \in \mathbb{N}$ such that for all $m \in \mathbb{N}$,

$$\pi_n(P) = \pi_{n+m}(P) = P.$$

The inclusion relation on threads in BTA is the partial ordering generated by the following two clauses:

- For all $P \in \mathrm{BTA}$, $\mathsf{D} \sqsubseteq P$.
- For all $P, P', Q, Q' \in \mathrm{BTA}$ and $a \in \mathcal{A}$, if $P \sqsubseteq P'$ and $Q \sqsubseteq Q'$ then $P \trianglelefteq a \trianglerighteq Q \sqsubseteq P' \trianglelefteq a \trianglerighteq Q'$.

BTA has a completion BTA^∞ which also comprises the infinite threads. BTA^∞ is the cpo consisting of all projective sequences. We define,

$$\mathrm{BTA}^\infty = \{(P_n)_{n\in\mathbb{N}} \mid \forall n \in \mathbb{N}(P_n \in \mathrm{BTA} \wedge \pi_n(P_{n+1}) = P_n)\}.$$

Now $(P_n)_{n\in\mathbb{N}} = (Q_n)_{n\in\mathbb{N}}$ if $P_n = Q_n$ for all $n \in \mathbb{N}$. Furthermore we overload notation and define,

$$\begin{aligned} \mathsf{D} &= (\mathsf{D}, \mathsf{D}, \dots), \\ \mathsf{S} &= (\mathsf{D}, \mathsf{S}, \mathsf{S}, \dots), \\ (P_n)_{n\in\mathbb{N}} \trianglelefteq a \trianglerighteq (Q_n)_{n\in\mathbb{N}} &= (R_n)_{n\in\mathbb{N}}, \text{with } \begin{cases} R_0 = \mathsf{D}, \\ R_{n+1} = P_n \trianglelefteq a \trianglerighteq Q_n. \end{cases} \end{aligned}$$

This definition also shows how all elements of BTA have a counterpart in BTA^∞. The projective sequence corresponding to a thread $P \in \mathrm{BTA}$ is $(\pi_n(P))_{n\in\mathbb{N}}$.

The set $\textsc{res}(P)$ of *residual threads* of P has the following inductive definition:

$$P \in \textsc{res}(P), \qquad Q \trianglelefteq a \trianglerighteq R \in \textsc{res}(P) \implies Q \in \textsc{res}(P) \wedge R \in \textsc{res}(P). \tag{2.1}$$

Depending on the execution environment a residual thread may be "reached" by performing zero or more actions.

A thread P is *regular* if $\textsc{res}(P)$ is finite. Regular threads are also called *finite state threads*. Every element of $\textsc{res}(P)$ is a state. We write $\mathrm{BTA}^{\mathrm{reg}} \subset \mathrm{BTA}^\infty$ for the set of regular threads.

A *finite linear recursive specification* over BTA^∞ is a set of equations

$$x_i = t_i$$

for $i \in I$ with I a finite index set, variables x_i and all t_i terms of the form S, D or $x_i \trianglelefteq a \trianglerighteq x_j$ with $j, k \in I$ and $a \in \mathcal{A}$. $P \in \mathrm{BTA}^{\mathrm{reg}}$ iff P is the solution of a finite recursive specification (see Theorem 1 of [BP09a]).

[2] In this thesis we will use the convention that $\mathbb{N}$ is the set of all natural numbers, including 0. $\mathbb{N}^+ = \mathbb{N} - \{0\}$. The integers are denoted $\mathbb{Z}$.

2.2 Program Algebra: PGA

A program can be viewed as a single-pass instruction sequence. That is, a program is a finite or infinite sequence of instructions which is executed from left to right such that every individual instruction is executed at most once—it is either executed or skipped. Single-pass instruction sequences are the main concept underlying PGA [PvdZ06, BL02]. Given an (implicit) set $\mathcal{A}$ of actions, PGA terms are constructed by concatenating instructions from the set $\mathfrak{I}$, defined as,

$$\mathfrak{I} = \bigcup_{a \in \mathcal{A}} \{a, +a, -a\} \cup \bigcup_{k \in \mathbb{N}} \{\#k\} \cup \{!\}.$$

The instructions in $\mathfrak{I}$ are called *primitive instructions*. Let us informally define their behavior (note that $a \in \mathcal{A}$ and $k \in \mathbb{N}$):

- a is a *basic instruction*. It instructs the execution environment to perform action a. The boolean reply returned by the environment is disregarded.
- $+a$ is a *positive test instruction*. Like a, it instructs execution of action a. However, only if the execution environment returns `true` will the instruction to its immediate right be executed. Otherwise this instruction is skipped and execution proceeds at the next instruction.
- $-a$ is a *negative test instruction*. This is the dual of the positive test instruction, in the sense that it skips the next instruction iff the environment returns `true` after performing action a.
- $\#k$ is a *forward jump instruction*. This instruction transfers execution to the kth instruction to its right (i.e., $k - 1$ instructions are skipped). Note that $\#0$ instructs the indefinite repetition of this instruction. Hence the behavior of $\#0$ is identified with deadlock.
- $!$ is the *termination instruction*. It causes successful termination of the program.

The set of PGA terms is denoted $\boldsymbol{P}$. PGA terms are constructed from primitive instructions using the binary *concatenation* operator $\_;\_$ and the unary *repetition* operator $\_^\omega$. That is, $\boldsymbol{P}$ is the smallest superset of $\mathfrak{I}$ that is closed under concatenation and repetition. Thus, for all $X, Y \in \boldsymbol{P}$, also $X; Y \in \boldsymbol{P}$ and $X^\omega \in \boldsymbol{P}$. Examples of PGA terms include:

$$a, \qquad +b; \#3, \qquad (\#3; a; b)^\omega, \qquad -c; -c; (-a)^\omega. \tag{2.2}$$

2.2.1 First Canonical Form

We define $X^1 = X$ and $X^{n+1} = X; X^n$, for all $n \in \mathbb{N}$. Using this notation, PGA defines the following four axioms for all $X, Y, Z \in \boldsymbol{P}$:

$$(X; Y); Z = X; (Y; Z) \tag{PGA1}$$
$$(X^n)^\omega = X^\omega \tag{PGA2}$$
$$X^\omega; Y = X^\omega \tag{PGA3}$$
$$(X; Y)^\omega = X; (Y; X)^\omega \tag{PGA4}$$

These four axioms define *instruction sequence congruence*. Instruction sequence congruent PGA expressions execute exactly the same instructions and are thus behaviorally equivalent. In the remainder of this thesis instruction sequence congruent PGA terms are identified.

(PGA1) states that concatenation is associative. Using (PGA2) and (PGA4) we derive that $X^\omega = X; X^\omega$ for all $X \in \boldsymbol{P}$. Furthermore, using (PGA1)–(PGA4) every PGA term can be rewritten to one of the following two forms:

1. X, where X does not contain the repetition operator, or
2. $X; Y^\omega$, with X and Y not containing the repetition operator.

Any PGA term in one of these two forms is said to be in *first canonical form*. The set $\boldsymbol{P_1} \subset \boldsymbol{P}$ contains exactly those PGA terms which are in first canonical form. The function $\textsc{fst}\colon \boldsymbol{P} \to \boldsymbol{P_1}$ converts any given PGA term to a first canonical form. Let $X_1, X_2, Y_1, Y_2 \in \boldsymbol{P}$, such that X_1 and X_2 do not contain repetition. Then FST can be defined such that,

$$\begin{aligned} \textsc{fst}(Y_1^\omega) &= \textsc{fst}(Y_1; Y_1^\omega) & \qquad \textsc{fst}(X_1) &= X_1 \\ \textsc{fst}(Y_1^\omega; Y_2) &= \textsc{fst}(Y_1^\omega) & \qquad \textsc{fst}(X_1; X_2^\omega) &= X_1; X_2^\omega \\ \textsc{fst}(X_1; Y_1^\omega; Y_2) &= \textsc{fst}(X_1; Y_1^\omega) \end{aligned}$$

It is not hard to see that FST is total and makes use only of (PGA1)–(PGA4).

2.2.2 Second Canonical Form

Another congruence relation defined on PGA terms is *structural congruence*. It is defined using the following four axioms which are concerned with chained jump instructions in PGA terms in first canonical form:

$$\#n{+}1; u_1; \ldots; u_n; \#0 = \#0; u_1; \ldots; u_n; \#0, \tag{PGA5}$$

$$\#n{+}1; u_1; \ldots; u_n; \#m = \#n{+}m{+}1; u_1; \ldots; u_n; \#m, \tag{PGA6}$$

$$(\#k{+}n{+}1; u_1; \ldots; u_n)^\omega = (\#k; u_1; \ldots; u_n)^\omega, \tag{PGA7}$$

and,

$$\#n{+}m{+}k{+}2; u_1; \ldots; u_n; (v_1; \ldots; v_{m+1})^\omega = \#n{+}k{+}1; u_1; \ldots; u_n; (v_1; \ldots; v_{m+1})^\omega. \tag{PGA8}$$

Using (PGA1)–(PGA8) every PGA term in first canonical form can be rewritten to a structurally congruent PGA term without chained jump instructions (this also implies that the jump counter of jump instructions into and inside the repeating part of a PGA term is minimal). Such a term is said to be in *second canonical form*. As with first canonical forms, second canonical forms are not unique. However, any second canonical form $X; Y^\omega$ can be converted to an equivalent second canonical form $X'; Y'^\omega$ where X' and Y' are minimal. Then $X'; Y'^\omega$ *is* unique.

The set $\boldsymbol{P_2} \subset \boldsymbol{P_1}$ contains exactly those PGA terms which are in second canonical form. The function $\textsc{snd}\colon \boldsymbol{P} \to \boldsymbol{P_2}$ converts any PGA term to its minimal second canonical form. We do not provide an implementation here.

2.2.3 The Semantics of PGA

Every PGA term $X \in \boldsymbol{P}$ has uniquely defined behavior, in the form of some thread $T \in \mathrm{BTA}^{\mathrm{reg}}$. The *thread extraction operator* $|\_|_{\mathrm{PGA}}\colon \boldsymbol{P} \to \mathrm{BTA}^{\mathrm{reg}}$ yields this thread, for every PGA term. It is defined as,

$$|X|_{\mathrm{PGA}} = \begin{cases} a \circ \mathsf{D} & \text{if } X \in \{a, +a, -a\}, \\ a \circ |Y|_{\mathrm{PGA}} & \text{if } X = a; Y, \\ |Y|_{\mathrm{PGA}} \trianglelefteq a \trianglerighteq |\#2; Y|_{\mathrm{PGA}} & \text{if } X = +a; Y, \\ |\#2; Y|_{\mathrm{PGA}} \trianglelefteq a \trianglerighteq |Y|_{\mathrm{PGA}} & \text{if } X = -a; Y, \\ |Y|_{\mathrm{PGA}} & \text{if } X = \#1; Y, \\ |\#k{+}1; X|_{\mathrm{PGA}} & \text{if } X = \#k{+}2; u; Y, \\ \mathsf{D} & \text{if } X \in \{\#k, \#0; Y, \#k{+}2; u\}, \\ \mathsf{S} & \text{if } X \in \{!, !; Y\}, \end{cases} \tag{2.3}$$

Note that this definition does not explicitly mention the repetition operator. Instead it uses the notion that X is "unfolded" when needed—by means of (PGA4) and possibly (PGA2). Thread extraction on PGA terms requires one additional rule:

> If the equations in (2.3) can be applied infinitely often from left to right without ever yielding an action, then the extracted thread is D. (2.4)

Observe that (2.4) is only relevant for PGA terms which contain an infinite sequence of chained jump instructions. As such it is not applicable to second canonical forms.

Examples Let us apply the thread extraction operator $|\_|_{\mathrm{PGA}}$ to the example PGA terms of (2.2).

- The behavior of the term a can be derived in a single step according to (2.3):

$$|a|_{\mathrm{PGA}} = a \circ \mathsf{D}.$$

- $+b;\#3$ appears to be a more complicated example, but its behavior turns out to be equally simple:

$$|+b;\#3|_{\mathrm{PGA}} = |\#3|_{\mathrm{PGA}} \trianglelefteq b \trianglerighteq |\#2;\#3|_{\mathrm{PGA}} = \mathsf{D} \trianglelefteq b \trianglerighteq \mathsf{D} = b \circ \mathsf{D}.$$

- It turns out that the single pass instruction sequence $(\#3;a;b)^\omega$ does not perform any action, despite its infinite length:

$$|(\#3;a;b)^\omega|_{\mathrm{PGA}} = |(\#0;a;b)^\omega|_{\mathrm{PGA}} = |\#0;(a;b;\#0)^\omega|_{\mathrm{PGA}} = \mathsf{D}.$$

  Observe that the first step of this derivation applies (PGA7), followed by an application of (PGA4).

- Lastly, $-c;-c;(-a)^\omega$ produces infinite behavior. To determine its exact behavior, we start out with a couple of left-to-right applications of (2.3):

$$\begin{aligned}
|-c;-c;(-a)^\omega|_{\mathrm{PGA}} &= |\#2;-c;(-a)^\omega|_{\mathrm{PGA}} \trianglelefteq c \trianglerighteq |-c;(-a)^\omega|_{\mathrm{PGA}} \\
&= |(-a)^\omega|_{\mathrm{PGA}} \trianglelefteq c \trianglerighteq (|\#2;(-a)^\omega|_{\mathrm{PGA}} \trianglelefteq c \trianglerighteq |(-a)^\omega|_{\mathrm{PGA}}) \\
&= |(-a)^\omega|_{\mathrm{PGA}} \trianglelefteq c \trianglerighteq (|\#2;-a;(-a)^\omega|_{\mathrm{PGA}} \trianglelefteq c \trianglerighteq |(-a)^\omega|_{\mathrm{PGA}}) \\
&= |(-a)^\omega|_{\mathrm{PGA}} \trianglelefteq c \trianglerighteq (|(-a)^\omega|_{\mathrm{PGA}} \trianglelefteq c \trianglerighteq |(-a)^\omega|_{\mathrm{PGA}}) \\
&= |(-a)^\omega|_{\mathrm{PGA}} \trianglelefteq c \trianglerighteq c \circ |(-a)^\omega|_{\mathrm{PGA}}.
\end{aligned}$$

  At this stage the behavior of $-c;-c;(-a)^\omega$ has not been fully derived, as the thread corresponding to $(-a)^\omega$ still needs to be determined. This thread turns out to be infinite:

$$\begin{aligned}
|(-a)^\omega|_{\mathrm{PGA}} &= |-a;(-a)^\omega|_{\mathrm{PGA}} \\
&= |\#2;(-a)^\omega|_{\mathrm{PGA}} \trianglelefteq a \trianglerighteq |(-a)^\omega|_{\mathrm{PGA}} \\
&= |\#2;-a;(-a)^\omega|_{\mathrm{PGA}} \trianglelefteq a \trianglerighteq |(-a)^\omega|_{\mathrm{PGA}} \\
&= |(-a)^\omega|_{\mathrm{PGA}} \trianglelefteq a \trianglerighteq |(-a)^\omega|_{\mathrm{PGA}} \\
&= a \circ |(-a)^\omega|_{\mathrm{PGA}}.
\end{aligned}$$

  It follows that $|(-a)^\omega|_{\mathrm{PGA}}$ can be described by the recursive specification $Q = a \circ Q$. Now, equating $|-c;-c;(-a)^\omega|_{\mathrm{PGA}}$ with P, we see that the behavior of $-c;-c;(-a)^\omega$ is equals P_1, as described by the following linear recursive specification:

$$P_1 = P_3 \trianglelefteq c \trianglerighteq P_2, \qquad P_2 = P_3 \trianglelefteq c \trianglerighteq P_3, \qquad P_3 = P_3 \trianglelefteq a \trianglerighteq P_3.$$

  (A shorter notation would be $P = Q \trianglelefteq c \trianglerighteq c \circ Q$, $Q = a \circ Q$.)

Proposition 2.1. *Each thread definable in PGA is regular, and each regular thread can be expressed in PGA.*

Proof. See e.g. Proposition 2 in [PvdZ06]. Alternatively, the result follows from the following two observations:

- The code semigroup C introduced in Chapter 3 characterizes the regular threads (see Proposition 3.1).
- There exist total behavior preserving mappings from PGA to C and vice versa (see §5.2 and §5.1, respectively). □

2.3 Finite Instruction Sequences and Code Semigroups

In PGA each instruction is executed at most once and the repetition operator $\_^\omega$ is used to construct infinite sequences of instructions. The instruction sequence semigroups introduced in the following chapters, on the other hand, represent only finite instruction sequences in which instructions can be executed multiple times in any order. This section introduces some relevant notions and terminology in preparation of the introduction of concrete code semigroups in Chapter 3 and Chapter 4.

2.3.1 Finite Instruction Sequences

Consider a non-empty instruction set $\mathcal{I}$ and an associative binary operation $\_;\_$ on $\mathcal{I}$. We will call $\_;\_$ the *concatenation operator*. Instructions can be concatenated, thereby yielding finite *instruction sequences* (*inseqs*) of arbitrary length. For all $n \in \mathbb{N}^+$, let

$$\mathcal{I}^1 = \mathcal{I}, \qquad \mathcal{I}^{n+1} = \{X; u \mid X \in \mathcal{I}^n, u \in \mathcal{I}^1\}.$$

Then $\mathcal{I}^n$ is the set of instruction sequences of length n. We define

$$\mathcal{I}^+ = \bigcup_{n \in \mathbb{N}^+} \mathcal{I}^n.$$

$\mathcal{I}^+$ contains all finite, non-empty (length greater than zero) sequences of $\mathcal{I}$-instructions. X is an *$\mathcal{I}$-inseq* iff $X \in \mathcal{I}^+$. An $\mathcal{I}$-inseq will also be called an *$\mathcal{I}$-expression*. We call $\ell\colon \mathcal{I}^+ \to \mathbb{N}^+$ the *length function*, and it is defined such that $\ell(X) = n$ iff $X \in \mathcal{I}^n$.

Concatenation is an associative operation, thus $(X;Y);Z = X;(Y;Z)$ for arbitrary $X, Y, Z \in \mathcal{I}^+$. Parentheses will therefore usually be omitted, and we write $X;Y;Z$. Note also, that it trivially follows that for arbitrary $n, m \geq 1$,

$$\mathcal{I}^{n+m} = \{X;Y \mid X \in \mathcal{I}^n, Y \in \mathcal{I}^m\}.$$

For convenience, we will write $\mathcal{I}^{\leq n}$ for the set of all $\mathcal{I}$-expressions up to length n. Likewise $\mathcal{I}^{\geq n}$ contains all $\mathcal{I}$-expressions of length n or greater. That is,

$$\mathcal{I}^{\leq n} = \{X \in \mathcal{I}^+ \mid \ell(X) \leq n\}, \qquad \mathcal{I}^{\geq n} = \{X \in \mathcal{I}^+ \mid \ell(X) \geq n\}.$$

For all $i \in \mathbb{N}^+$, we define auxiliary functions $\sigma_i\colon \mathcal{I}^{\geq i} \to \mathcal{I}$ which return the ith instruction in a given $\mathcal{I}$-inseq. That is, if $X = u_1; u_2; \ldots; u_n$, then $\sigma_i(X) = u_i$ for all $1 \leq i \leq n$. We define $i =_X j$ iff $\sigma_i(X) = \sigma_j(X)$. Clearly $=_X$ is an equivalence relation.

Next, for all $X \in \mathcal{I}^+$ and $U \subseteq \mathcal{I}$ we define $U(X) = \{i \mid \sigma_i(X) \in U\}$. In other words, $U(X)$ contains the positions in the $\mathcal{I}$-inseq X of instructions contained in U.

It will sometimes prove convenient to regard an inseq X as a set whose elements are the distinct instructions contained in X. So for any $X \in \mathcal{I}^+$ we write $u \in X$ to indicate that $\sigma_i(X) = u$ for some i. $X \cap S$ and $X \cup S$ are defined as one would expect them to be (note that S can be a set or another inseq).

About Notation Let $X \in \mathcal{I}^+$ be an instruction sequence. Throughout this thesis we will write X^k for k concatenations of X. That is,

$$X^1 = X, \qquad X^{n+1} = X; X^n.$$

What about X^0? Our definition of an instruction sequence explicitly excludes the empty sequence: an $\mathcal{I}$-expression will always contain at least one instruction. Still, within some contexts it will prove convenient to talk about X^k for *any* $k \in \mathbb{N}$. Throughout this thesis we will only write X^0 as part of sequences which, as a whole, are guaranteed to be non-empty, and are as such contained in $\mathcal{I}^+$ (i.e., the set of proper instruction sequences).

2.3.2 Code Semigroups

Given some instruction set $\mathcal{I}$, every inseq $X \in \mathcal{I}^+$ is constructed by concatenation of a finite number of elements in $\mathcal{I}$. Hence $\mathcal{I}$ *generates* $\mathcal{I}^+$, denoted $<\mathcal{I}> = \mathcal{I}^+$. $\mathcal{I}^+$ is closed under the associative binary operation $\_;\_$ and as such $\mathcal{I}^+$ is a semigroup with respect to $\_;\_$. Clearly every instruction set $\mathcal{I}$ gives rise to a semigroup $(<\mathcal{I}>, \_;\_)$. We will call such a semigroup an *instruction sequence semigroup* or simply *code semigroup*. For an introduction to semigroup theory we refer to [CP61].

About Notation Let B refer to some code semigroup. Then we write $\mathcal{I}_B$ for the instruction set of B. $\mathcal{I}_B^n$ denotes the $\mathcal{I}_B$-inseqs of length n, and $\mathcal{I}_B^+$ contains all $\mathcal{I}_B$-expressions. Hence we write $B = (\mathcal{I}_B^+, \_;\_)$. When no confusion can arise, $\mathcal{I}_B$-instructions and $\mathcal{I}_B$-inseqs may simply be referred to as B-instructions and B-inseqs (B-expressions), respectively. Whenever B is referred to as a set instead of a semigroup, it is identified with $\mathcal{I}_B^+$. That is, B stands for all well-formed B-expressions. Likewise $\mathcal{I}_B^+$ may be referred to as a semigroup, in which case it is identified with B.

Subsemigroups Let A and B be two semigroups with respect to some operator $\bullet$, such that $A \subseteq B$. Then A is a subsemigroup of B. Equivalently, if $(B, \bullet)$ is a semigroup and $A \subseteq B$ such that $a, b \in A$ implies $a \bullet b \in A$, then A is a subsemigroup of B. Note that the intersection of any subsemigroups of B is either empty or itself a subsemigroup of B.[3]

Given an instruction set $\mathcal{I}$ we can take a subset of these instructions, $\mathcal{I}' \subseteq \mathcal{I}$. Observe that the semigroup $<\mathcal{I}'>$ is a strict subsemigroup of $<\mathcal{I}>$. We will define plenty of such subsemigroups later in this thesis.

Semigroup Homomorphisms Consider two code semigroups A and B, and a function $f\colon \mathcal{I}_A^+ \to \mathcal{I}_B^+$. Then f is a mapping between instruction sequences. A significant part of this thesis describes mappings between distinct code semigroups. Most of these mappings are homomorphisms.

In general, a function $f\colon A \to B$ is a homomorphism between semigroups $(A, \bullet)$ and $(B, *)$ iff $f(x \bullet y) = f(x) * f(y)$, for all $x, y \in A$. It is easy to see that f only needs to be defined explicitly on elements of A's generating set. If $<G> = A$, then for all $a \in A - G$ it is the case that $a = g_0 \bullet g_1 \bullet \cdots \bullet g_{n+1}$, for some $n \in \mathbb{N}$ and $g_0, g_1, \ldots, g_{n+1} \in G$, and hence $f(a) = f(g_0) * f(g_1) * \cdots * f(g_{n+1})$ by definition. In the specific case of code semigroups this implies that a homomorphic function only needs to be defined explicitly on individual instructions.

2.3.3 Instruction Sequence Semantics

It is the ability to be *executed* that sets instruction sequences apart from sequences of arbitrary mathematical objects. Execution of an instruction sequence leads to (possibly

[3] We will not consider the empty semigroup.

unobservable) behavior. Thus, for a sequence of objects to be called an instruction sequence, it must be ascribed a semantics, such that its behavior upon execution is defined.

This thesis will use basic thread algebra to that end. This allows us to define the semantics of the semigroup C as in [BP09a] and provides an easy way to compare the code semigroups introduced in this thesis to PGA on a syntactical as well as a semantic level.

In the tradition of PGA instructions are viewed as atomic program components: at any stage during the execution of a program at most one instruction is "active" (i.e., being executed).[4] We will define the behavior of individual instructions based on their position i within an instruction sequence X. Execution of an individual instruction may or may not cause an action to be performed, after which control of execution is transferred to another position in X. Then, given the position of the first instruction to be executed, the semantics of the instruction sequence as a whole follows naturally.

The first instruction to be executed is called the *initial* or *start instruction*. The leftmost and rightmost instruction of an inseq are obvious candidates to be designated as such, but given a specific instruction sequence X, execution can start at any position within X. Thus for all $X \in \mathcal{I}^+$ and $1 \leq i \leq \ell(X)$, the pair (i, X) can be identified with a certain thread, namely the thread which represents the behavior resulting from the execution of X starting with the ith instruction. Though not strictly necessary, for any invalid instruction position i (i.e. $i < 1$ or $i > \ell(X)$) the pair (i, X) will be identified with some default thread D. Once D has been fixed, every pair $(i, X) \in \mathbb{Z} \times \mathcal{I}^+$ is identified with a certain thread T. Throughout this thesis we will consider only one value for D, namely D, i.e. deadlock.

In this way the *thread extraction operator* $|\_, \_| \colon \mathbb{Z} \times \mathcal{I}^+ \to \mathrm{BTA}^\infty$ specifies the semantics of a semigroup $\mathcal{I}^+$. For convenience we will usually write $|X|^i$ instead of $|i, X|$, but this is merely a notational matter. For any $X \in \mathcal{I}^+$, the thread describing the behavior of X if executed starting from the leftmost instruction is called its *left behavior*, written $|X|^\rightarrow = |X|^1$. Likewise $|X|^\leftarrow = |X|^{\ell(X)}$ is called the *right behavior* of X, meaning the behavior of X if executed starting from the rightmost instruction.

Once specific code semigroups have been defined—along with suitable thread extraction operators—it becomes possible to analyze their expressiveness. Given equally expressive code semigroups A and B one can define mappings between them, such that the behavior of any inseq X in the domain is in some way reflected by the behavior of the corresponding inseq Y to which it is mapped in the codomain. Similar mappings can also be defined from a semigroup A onto itself.

Definition 2.2. Let A and B be two code semigroups on which the thread extraction operators $|\_, \_|_A \colon \mathbb{Z} \times \mathcal{I}_A^+ \to \mathrm{BTA}^\infty$ and $|\_, \_|_B \colon \mathbb{Z} \times \mathcal{I}_B^+ \to \mathrm{BTA}^\infty$ are defined, respectively. Consider arbitrary $X \in \mathcal{I}_A^+$ and $Y \in \boldsymbol{P}$ and three mappings $f \colon \mathcal{I}_A^+ \to \mathcal{I}_B^+$, $g \colon \mathcal{I}_A^+ \to \boldsymbol{P}$ and $h \colon \boldsymbol{P} \to \mathcal{I}_A^+$. Then,

- f is *left behavior preserving* if $|X|_A^\rightarrow = |f(X)|_B^\rightarrow$.
- f is *right behavior preserving* if $|X|_A^\leftarrow = |f(X)|_B^\leftarrow$.
- f is *left-right behavior preserving* if it is both left and right behavior preserving.
- f is *behavior preserving* if it is left or right behavior preserving.
- f is *left uniformly behavior preserving* if there exists some $b \in \mathbb{N}^+$ such that $|X|_A^i = |f(X)|_B^{b(i-1)+1}$ for all $i \in \mathbb{Z}$. Observe that every left uniformly behavior preserving mapping is left behavior preserving.

[4] One could draw a parallel with the program counter as found in central processing units (CPUs), which holds the memory address of the instruction that is currently executed (or the instruction which is to be executed next, depending on the architecture).

- f is *right uniformly behavior preserving* if there exists some $b \in \mathbb{N}^+$ such that $|X|_A^i = |f(X)|_B^{bi}$ for all $i \in \mathbb{Z}$. Observe that every right uniformly behavior preserving mapping is right behavior preserving.
- f is *left-right uniformly behavior preserving* if it is both left and right uniformly behavior preserving.
- f is *uniformly behavior preserving* if it is left or right uniformly behavior preserving.
- g is *behavior preserving* if $|X|_A^{\rightarrow} = |g(X)|_{\mathrm{PGA}}$.
- h is *behavior preserving* if $|Y|_{\mathrm{PGA}} = |h(Y)|_A^{\rightarrow}$.[5]

A behavior preserving mapping will also be called a *translation* because it preserves the meaning of the original (single pass) instruction sequence.

This concludes the preliminaries. We are now ready to introduce the code semigroup C in the next chapter.

[5] A more general definition would be that h is behavior preserving if there exists a function $t\colon \boldsymbol{P} \to \mathbb{Z}$ such that $|Y|_{\mathrm{PGA}} = |h(Y)|_A^{t(Y)}$, but this definition suffices for our purposes.

Chapter 3

C Instruction Sequences

The previous chapter introduced PGA as a means to describe programs and BTA as a means to describe their behavior. It then introduced an alternative representation of program objects, namely strictly finite instruction sequences, as opposed to PGA's infinite single-pass instruction sequences. Upon specifying an instruction set $\mathcal{I}$ the set of finite instruction sequences generated by concatenating elements of $\mathcal{I}$ forms a semigroup. This chapter introduces one such semigroup and its semantics.

C was first described in [BP09a]. C is a code semigroup without directional bias: execution of a C-inseq can start at the leftmost instruction (the natural choice for most people in Western society), but may just as well start at the rightmost instruction. In fact, given some instruction sequence X, any position within X can be designated as starting position.

This chapter is built up as follows: §3.1 will introduce C's instruction set and provide some basic examples of C-expressions. It will also motivate the inclusion in the instruction set of an instruction which upon execution will cause deadlock. Next, §3.2 formalizes the semantics of C-expressions using thread algebra. Based on this, §3.3 introduces some accessibility relations on instruction positions which will be used throughout this thesis. Lastly, §3.4 briefly discusses a small syntactic and semantic variation on C.

3.1 The Instruction Set

Given a set $\mathcal{A}$ of actions, C defines basic instructions $\mathfrak{B}$, positive test instructions $\mathfrak{P}$, negative test instructions $\mathfrak{N}$ and relative jumps $\mathfrak{J}$:

$$\mathfrak{B} = \bigcup_{a\in\mathcal{A}} \{/a, \backslash a\}, \qquad \mathfrak{J} = \bigcup_{k\in\mathbb{N}^+} \{/\#k, \backslash\#k\},$$
$$\mathfrak{P} = \bigcup_{a\in\mathcal{A}} \{+/a, +\backslash a\}, \qquad \mathfrak{N} = \bigcup_{a\in\mathcal{A}} \{-/a, -\backslash a\}.$$

$\mathcal{A}$ is a parameter to C which is often kept implicit. Additionally, C has an abort instruction $\#$ and termination instruction !. Instructions with a backward slash are called *left oriented* or *backward* instructions; those with a forward slash are called *right oriented* or *forward* instructions. Instructions with a left (right) orientation are also said to have a left (right) *directionality*. Formally, $C = (\mathcal{I}_C^+, \_;\_)$, with the set of all C-expressions $\mathcal{I}_C^+$ generated by C's instruction set $\mathcal{I}_C$, defined as

$$\mathcal{I}_C = \mathfrak{B} \cup \mathfrak{P} \cup \mathfrak{N} \cup \mathfrak{J} \cup \{\#, !\}.$$

Let $a, b, c \in \mathcal{A}$. Then examples of C-expressions are

$$/a, \qquad /a; +/a; !; \backslash\#3, \qquad +/b; -/c; -\backslash c, \qquad \backslash\#2; -\backslash c. \tag{3.1}$$

Each C-inseq has a semantics. Before we formalize this, it will prove convenient to informally describe the meaning of some of the instructions:

$/a$ is a *forward basic instruction*. It causes execution of the action a, after which the instruction to its right is executed, if it exists. Otherwise deadlock occurs. Note that the boolean reply resulting from a's execution is ignored.

$+/a$ is a *forward positive test instruction*. Action a is executed. If its boolean reply is `true`, then the instruction immediately to its right is executed. On `false`, however, this instruction is skipped, and execution proceeds at the second instruction to its right. If no such instruction exists, deadlock follows.

$-/a$ is a *forward negative test instruction*. $-/a$ mirrors the behavior of $+/a$, in the sense that the effect of the replies `true` and `false` is reversed.

$/\#k$ is a *forward jump instruction*. It causes execution of the instruction k positions to its right, if such instruction exists. Otherwise deadlock will follow.

$\#$ is the *abort instruction*. Execution of this instruction causes deadlock.

$!$ is the *termination instruction*. It causes the program to halt successfully.

The instructions $\backslash a$, $+\backslash a$, $-\backslash a$ and $\backslash\#k$ are the *backward* versions of $/a$, $+/a$, $-/a$ and $/\#k$, respectively, in the sense that they have a right-to-left instead of a left-to-right orientation. For example, execution of $\backslash a$ results in action a, after which the instruction to its *left* is executed (if such instruction exists).

A jump instruction $/\#k$ or $\backslash\#k$ has *jump counter* k and performs a jump of *distance* k instructions. $/\#k$ or $\backslash\#k$ are said to be *relative jumps*. The function $\delta\colon \mathfrak{J} \to \mathbb{N}^+$ returns the jump counter of a given jump instruction (e.g. $\delta(/\#6) = 6$).

We define $\mathcal{I}_C^{\rightarrow} \subset \mathcal{I}_C$ to be the set forward instructions. Likewise, $\mathcal{I}_C^{\leftarrow} \subset \mathcal{I}_C$ denotes the set of backward instructions. Formally:[1]

$$\mathcal{I}_C^{\rightarrow} = \bigcup_{a\in\mathcal{A}} \{/a, +/a, -/a\} \cup \bigcup_{k\in\mathbb{N}^+} \{/\#k\},$$
$$\mathcal{I}_C^{\leftarrow} = \bigcup_{a\in\mathcal{A}} \{\backslash a, +\backslash a, -\backslash a\} \cup \bigcup_{k\in\mathbb{N}^+} \{\backslash\#k\}.$$

The sets $\mathfrak{B}^{\rightarrow} = \mathfrak{B} \cap \mathcal{I}_C^{\rightarrow}$ and $\mathfrak{B}^{\leftarrow} = \mathfrak{B} \cap \mathcal{I}_C^{\leftarrow}$ denote the forward and backward oriented basic instructions, respectively. Likewise for $\mathfrak{P}$, $\mathfrak{N}$ and $\mathfrak{J}$. Note that with the exception of the abort instruction $\#$ and the termination instruction $!$, every C-instruction has a direction, which is either forward or backward, but not both. That is, $\mathcal{I}_C^{\rightarrow} \cap \mathcal{I}_C^{\leftarrow} = \emptyset$ and $\mathcal{I}_C^{\rightarrow} \cup \mathcal{I}_C^{\leftarrow} \cup \{\#, !\} = \mathcal{I}_C$. We write $u \sim v$ if instructions u and v have the same direction (or no direction). $\sim \subset \mathcal{I}_C \times \mathcal{I}_C$ is the *directionality relation*. It is clearly an equivalence relation.

Examples These informal definitions of the meaning of each instruction allow us to verbally describe the meaning of the example C-expressions of (3.1), provided that we agree upon which instruction is the first to be executed. Since this thesis is written in English, which has an obvious left-to-right bias, we will designate the leftmost instruction to be the initial instruction. Thus we will informally describe these inseq's left behavior.

- $/a$: Performs action a, after which deadlock occurs.
- $/a; +/a; !; \backslash\#3$: Performs action a twice in a row. If the second action yields a positive reply, then the program terminates. Otherwise it starts all over.

[1] Note, again, that the set $\mathcal{A}$ of actions is an implicit parameter for C (and thereby for $\mathcal{I}_C$, $\mathcal{I}_C^{\rightarrow}$ and $\mathcal{I}_C^{\leftarrow}$).

- $+/b; -/c; -\backslash c$: Performs action b. If this yields the reply `true`, then action c will be performed, as specified by the second instruction. Here, a positive reply causes deadlock and a negative reply causes the third instruction to be executed. If the action b yields `false` then the third instruction will also be executed. The action c as performed by the third instruction causes execution to continue at either the first or second instruction, depending on whether it yields is a positive or negative reply, respectively.[2]

- $\backslash\#2; -\backslash c$: Does not perform any action. Execution of this program immediately causes deadlock, since the first instruction jumps outside of the inseq.

3.1.1 The Case for an Explicit Abort Instruction

A draft version of the original paper on C [BP09b] provided a definition of the semigroup C which differs slightly from the one that was published in [BP09a] (which is introduced in the previous section). Let us refer to the semigroup as it was introduced in [BP09b] by the name C'.

The instruction set $\mathcal{I}_{C'}$ did not contain an explicit abort instruction. It did however contain two other instructions which C lacks: $/\#0$ and $\backslash\#0$, both of which signify a jump of distance zero[3]. That is, $C' = (\mathcal{I}^+_{C'}, \_;\_)$, with

$$\mathcal{I}_{C'} = \{/\#0, \backslash\#0\} \cup \mathcal{I}_C - \{\#\}.$$

Since $/\#0$ and $\backslash\#0$ are under all circumstances behaviorally indistinguishable, C' had an extra axiom (aside from the obvious axiom which states that concatenation is associative) which stated that no distinction is made between forward and backward jumps of distance 0:

$$/\#0 = \backslash\#0.$$

A jump of distance 0 is not really a jump at all and it is rather meaningless to talk about the direction of such a jump. Semantically both $/\#0$ and $\backslash\#0$ signify deadlock. Moreover, the introduction of two distinct but equivalent instructions allows for the definition of a mapping f on $\mathcal{I}^+_{C'}$ such that $X = Y$ while $f(X) \neq f(Y)$.

It was therefore argued that C' should only contain jumps $/\#k$ and $\backslash\#k$ for $k > 0$, together with a single non-directional abort instruction $\#$, thereby eliminating the need for the axiom $/\#0 = \backslash\#0$ while retaining a single instruction with essentially the same behavior as $/\#0$ and $\backslash\#0$. This chain of reasoning naturally lead to the definition of an alternative semigroup, the one introduced in [BP09a] and the previous section under the name C.[4]

Execution of the abort instruction has the same effect as an attempt to transfer execution to a non-existing instruction. Since every instruction sequence is finite, one can take any inseq $X \in \mathcal{I}^+_C$ and construct a behaviorally equivalent inseq X' by replacing every abort instruction with a jump to a position < 1 or $> \ell(X)$. Hence $\#$ does not increase C's expressiveness. Still, as we will later see, the abort instruction is a convenient addition to the instruction set.

[2] Compare the length of this description to that of the actual program, and it becomes apparent that natural language is not really suited to produce concise descriptions of program behavior. There is also the problem of the inherent ambiguity of natural language. Luckily basic thread algebra provides a concise and unambiguous alternative!

[3] The existence of these instructions was probably inspired by the #0 instruction as found in PGA.

[4] The introduction of the instruction # is not really a first. A similar instruction can be found in [BL00], where it is introduced as part of PGA. It must be noted though, that [BL00] ascribes a different semantics to #, namely *meaningless behavior*, than to #0, which produces *divergent behavior*. The latter notion coincides with what is referred to in this thesis as deadlock (D in basic thread algebra). BTA does not provide a constant to represent meaningless behavior. As mentioned in a footnote in [BL02], # was later dropped and should in hindsight be seen as an abbreviation for #0.

For completeness, we define two homomorphisms $f\colon \mathcal{I}^+_{C'} \to \mathcal{I}^+_C$ and $g\colon \mathcal{I}^+_C \to \mathcal{I}^+_{C'}$ which make the correspondence between C' and C explicit. They are defined on individual instructions u as follows:

$$f(u) = \begin{cases} \# & \text{if } u \in \{/\#0, \backslash\#0\}, \\ u & \text{otherwise,} \end{cases} \qquad g(u) = \begin{cases} /\#0 & \text{if } u = \#, \\ u & \text{otherwise.} \end{cases}$$

Now clearly $f \circ g$ is the identity function on C-expressions. The axiom $/\#0 = \backslash\#0$ ensures that likewise $g \circ f$ is an identity function on C'-expressions.

3.2 Semantics

As discussed in §2.3.3, C's semantics are defined using basic thread algebra. Thus any combination of start position i and inseq $X \in \mathcal{I}^+_C$ is assigned some thread $|i, X|_C$. Writing $|X|^i_C$ for $|i, X|_C$, the thread extraction operator $|\_, \_|_C\colon \mathbb{Z} \times \mathcal{I}^+_C \to \mathrm{BTA}^{\mathrm{reg}}$ is defined on all $i \in \mathbb{Z}$ and $X \in \mathcal{I}^+_C$ as,

$$|X|^i_C = \begin{cases} \mathsf{D} & \text{if } i < 1 \text{ or } i > \ell(X), \\ a \circ |X|^{i+1}_C & \text{if } \sigma_i(X) = /a, \\ |X|^{i+1}_C \trianglelefteq a \trianglerighteq |X|^{i+2}_C & \text{if } \sigma_i(X) = +/a, \\ |X|^{i+2}_C \trianglelefteq a \trianglerighteq |X|^{i+1}_C & \text{if } \sigma_i(X) = -/a, \\ |X|^{i+k}_C & \text{if } \sigma_i(X) = /\#k, \\ a \circ |X|^{i-1}_C & \text{if } \sigma_i(X) = \backslash a, \\ |X|^{i-1}_C \trianglelefteq a \trianglerighteq |X|^{i-2}_C & \text{if } \sigma_i(X) = +\backslash a, \\ |X|^{i-2}_C \trianglelefteq a \trianglerighteq |X|^{i-1}_C & \text{if } \sigma_i(X) = -\backslash a, \\ |X|^{i-k}_C & \text{if } \sigma_i(X) = \backslash\#k, \\ \mathsf{D} & \text{if } \sigma_i(X) = \#, \\ \mathsf{S} & \text{if } \sigma_i(X) = !. \end{cases} \tag{3.2}$$

In words, the thread $|X|^i_C$ describes the behavior resulting from the execution of the inseq X starting at the ith instruction. Recall that we defined $|X|^{\rightarrow}_C = |X|^1_C$ and $|X|^{\leftarrow}_C = |X|^{\ell(X)}_C$ to mean X's left and right behavior, respectively.

Examples We will apply thread extraction on the instruction sequences of (3.1) to determine their left as well as right behavior.

- The C-expression $/a$ consists of a single instruction, and as such its left and right behavior are equivalent:

$$|/a|^{\rightarrow}_C = |/a|^1_C = a \circ \mathsf{D}, \qquad |/a|^{\leftarrow}_C = |/a|^{\ell(/a)}_C = |/a|^1_C = a \circ \mathsf{D}.$$

- Let $X = /a; +/a; !; \backslash\#3$. The left behavior of this instruction sequence is infinite, as we have seen in §3.1. This is confirmed by several applications of equations in (3.2):

$$|X|^{\rightarrow}_C = |X|^1_C = a \circ |X|^2_C = a \circ (|X|^3_C \trianglelefteq a \trianglerighteq |X|^4_C) = a \circ (\mathsf{S} \trianglelefteq a \trianglerighteq |X|^1_C).$$

  Observe that $|X|^{\rightarrow}_C$ is recursively defined by the equation $P = a \circ (\mathsf{S} \trianglelefteq a \trianglerighteq P)$. As for the right behavior of X, we observe that $|X|^{\leftarrow}_C = |X|^{\ell(X)}_C = |X|^4_C = |X|^1_C = |X|^{\rightarrow}_C$.

- Let $X = +/b; -/c; -\backslash c$. Upon trying to extract its behavior, we see that

$$\begin{aligned}|X|^{\rightarrow}_C &= |X|^1_C \\ &= |X|^2_C \trianglelefteq b \trianglerighteq |X|^3_C \\ &= (|X|^4_C \trianglelefteq c \trianglerighteq |X|^3_C) \trianglelefteq b \trianglerighteq (|X|^1_C \trianglelefteq c \trianglerighteq |X|^2_C) \\ &= (\mathsf{D} \trianglelefteq c \trianglerighteq (|X|^1_C \trianglelefteq c \trianglerighteq |X|^2_C)) \trianglelefteq b \trianglerighteq (|X|^1_C \trianglelefteq c \trianglerighteq |X|^2_C).\end{aligned}$$

  The behavior is clearly infinite, and no single recursive equation can describe it. The following linear recursive specification does:

$$P_1 = P_2 \trianglelefteq b \trianglerighteq P_3, \qquad P_2 = P_4 \trianglelefteq c \trianglerighteq P_3, \qquad P_3 = P_1 \trianglelefteq c \trianglerighteq P_2, \qquad P_4 = \mathsf{D}.$$

  Now $|X|^{\rightarrow}_C = P_1$ and $|X|^{\leftarrow}_C = P_3$.

- Let $X = \backslash\#2; -\backslash c$. Then $|X|^{\rightarrow}_C = \mathsf{D}$ and $|X|^{\leftarrow}_C = c \circ \mathsf{D}$, because

$$\begin{aligned}|X|^{\rightarrow}_C &= |X|^1_C = |X|^{-1}_C = \mathsf{D} \\ |X|^{\leftarrow}_C &= |X|^2_C = |X|^0_C \trianglelefteq c \trianglerighteq |X|^1_C = \mathsf{D} \trianglelefteq c \trianglerighteq |X|^{-1}_C = \mathsf{D} \trianglelefteq c \trianglerighteq \mathsf{D} = c \circ \mathsf{D}\end{aligned}$$

Loops Without Activity C-inseqs may contain chained jump instructions which form a loop. The equations of (3.2) do not adequately handle this situation, as they do not assign any specific thread to the execution of such a loop. Hence we introduce an additional rule for the extraction of behavior from C instruction sequences:[5]

> If the equations in (3.2) can be applied infinitely often from left to right without ever yielding an action, then the extracted thread is D. (3.3)

As an example of the application of this rule, consider the C instruction sequence $X = /\#3; \backslash\#1; !; \backslash\#2; \#; +\backslash a$. Its left behavior is

$$|X|^{\rightarrow}_C = |X|^1_C = |X|^4_C = |X|^2_C = |X|^1_C = \mathsf{D}.$$

Here we derive that $|X|^1_C$, $|X|^4_C$ and $|X|^2_C$ equal D by means of three left-to-right applications of equations in (3.2) followed by application of (3.3). Indeed, the instructions at positions 1, 2 and 4 form a closed loop without any non-jump instructions. This example is also yet another demonstration of the fact that the left and right behavior of an inseq are in general not equivalent; the right behavior of X is,

$$|X|^{\leftarrow}_C = |X|^{\ell(X)}_C = |X|^6_C = |X|^5_C \trianglelefteq a \trianglerighteq |X|^4_C = \mathsf{D} \trianglelefteq a \trianglerighteq \mathsf{D} = a \circ \mathsf{D}.$$

Proposition 3.1. *Each thread definable in C is regular, and each regular thread can be expressed in C.*

Proof. Let $X \in \mathcal{I}^+_C$. Following (3.2) and (3.3) we have that for arbitrary $i \in [1, \ell(X)]$ one of the following is the case (for some $j, k \in \mathbb{Z}$):

$$|X|^i_C = \mathsf{S}, \qquad |X|^i_C = \mathsf{D}, \qquad |X|^i_C = |X|^j_C \trianglelefteq a \trianglerighteq |X|^k_C.$$

Let $[i]_X = \{j \in [1, \ell(X)] \mid |X|^i_C = |X|^j_C\}$ be an equivalence class of positions in X from which identical behavior can be extracted. Let Q be the corresponding quotient set of $[1, \ell(X)]$. Then for all $[i] \in Q$ we define,

$$P_{[i]} = \begin{cases} \mathsf{S} & \text{if } |X|^i_C = \mathsf{S}, \\ \mathsf{D} & \text{if } |X|^i_C = \mathsf{D}, \\ P_{[j]} \trianglelefteq a \trianglerighteq P_{[k]} & \text{if } |X|^i_C = |X|^j_C \trianglelefteq a \trianglerighteq |X|^k_C. \end{cases}$$

[5] This rule is near identical to the rule (2.4) which assigns a thread to infinite sequences of chained jump instructions in PGA.

Now for all $i \in [1, \ell(X)]$ the thread $|X|_C^i$ equals $P_{[i]}$, which is completely specified by the above linear equations and is thus regular.

Conversely, let $T \in \mathrm{BTA}^{\mathrm{reg}}$ be described by the linear equations $P_0 = t_1$, $P_2 = t_2$, ..., $P_{n-1} = t_{n-1}$. Then there exists an $X \in \mathcal{I}_C^+$ with $\ell(X) = 3n$ such that $P_i = |X|_C^{3i+1}$ and thus specifically $|X|_C^{\rightarrow} = P_0$. We construct X as follows: If $P_i = \mathsf{S}$ then set $\sigma_{3i+1}(X) = !$. If $P_i = \mathsf{D}$ then set $\sigma_{3i+1}(X) = \#$. Otherwise $P_i = P_j \trianglelefteq a \trianglerighteq P_k$, thus we set $\sigma_{3i+1}(X) = +/a$. $\sigma_{3i+2}(X)$ and $\sigma_{3i+3}(X)$ are jump instructions to positions $3j+1$ and $3k+1$, respectively. Positions in X for which no instruction has been specified can be assigned an arbitrary instruction. □

3.3 The Reachability of Instructions

If the equations in (3.2) are read strictly from left to right, then they define for a given inseq X and an arbitrary instruction at position i in X which action said instruction performs (if any) and at which program position(s) j execution may proceed. Let us define this relation between program positions as follows:

Definition 3.2. Let $X \in \mathcal{I}_C^+$. Then the *accessibility relation* $\rightarrow_X \subset \mathbb{Z}^2$ of X is defined as:

$$\begin{aligned} i \rightarrow_X j \iff & \text{for some } a \in \mathcal{A} \text{ and } k \in \mathbb{Z},\ |X|_C^i \text{ equals one of} \\ & \{|X|_C^j,\ a \circ |X|_C^j,\ |X|_C^j \trianglelefteq a \trianglerighteq |X|_C^k,\ |X|_C^k \trianglelefteq a \trianglerighteq |X|_C^j\} \\ & \text{according to a single left-to-right application of an equation in (3.2).} \end{aligned}$$

That is, $i \rightarrow_X j$ iff execution may continue at position j right after the instruction at position i has been executed. We then call i the *source position* and $\sigma_i(X)$ the *source instruction*. Likewise j and $\sigma_j(X)$ are the *target position* and *target instruction*, respectively.

As usual, $\rightarrow_X^+$ denotes the transitive closure of the relation $\rightarrow_X$. Likewise $\rightarrow_X^*$ is its reflexive and transitive closure.

Definition 3.3. Let $X \in \mathcal{I}_C^+$. A program position j is *reachable* from position i in X if $i \rightarrow_X^* j$.[6] The set $\mathcal{R}_{X,i} = \{j \mid i \rightarrow_X^* j\}$ contains i and all positions reachable from i in X. It's complement $\overline{\mathcal{R}_{X,i}} = \mathbb{Z} - \mathcal{R}_{X,i}$ naturally contains those positions which are unreachable from i. Note that $\mathcal{R}_{X,i}$ may include "invalid" program positions, i.e. positions outside of X.

Definition 3.4. The set $\mathcal{E}_X = \{i \in [1, \ell(X)] \mid i \rightarrow_X j, j \notin [1, \ell(X)]\}$ contains the *exit positions* of X. That is, execution of an instruction at some position in $\mathcal{E}_X$ may cause a position outside X to be "reached".

Proposition 3.5. *Every regular thread can be described by a C instruction sequence in which every instruction is reachable from the start instruction.*

Proof. Consider arbitrary $T \in \mathrm{BTA}^{\mathrm{reg}}$, $X \in \mathcal{I}_C^+$ and $i \in [1, \ell(X)]$ such that $|X|_C^i = T$. If $\overline{\mathcal{R}_{X,i}} \cap [1, \ell(X)] = \emptyset$, then T, X and i meet the requirements. Otherwise, randomly select some unreachable position $j \in \overline{\mathcal{R}_{X,i}} \cap [1, \ell(X)]$.

If the jth instruction is removed from X, then the jump counter of any jump instruction which jumps over position j should be reduced by one, so as to ensure that its target instruction remains the same. This is possible since said jump counter must be at least 2. We do not have to be concerned with any other instruction which can transfer control of execution to or over position j; such an instruction must itself not be reachable (because position j isn't) and has as such no effect on X's behavior.

The result of removing the instruction at position j from X is an inseq X' such that either $|X'|_C^{i-1} = T$ or $|X'|_C^i = T$, depending on whether $j < i$ or $j > i$, respectively. This process can be repeated until all unreachable instructions are removed. □

[6] Note that every instruction is reachable from itself. This is somewhat unconventional, but convenient for our purposes.

3.4 A Small Variation on C

For each $a \in \mathcal{A}$, C provides four test instructions: $+/a$, $-/a$, $+\backslash a$ and $-\backslash a$. Semantically speaking the first two of these have immediate counterparts in PGA: $+a$ and $-a$. The latter two are backward versions of the former two, and thus are indirectly based on (or even inspired by) the PGA test instructions as well.

C's lack of directional bias allows for a different semantics for test instructions, though; one that is instead inspired by the postconditional composition operator as found in basic thread algebra. Consider the following two instructions:

- $+a$ is the *positive test instruction*. It performs action a. If the environment returns `true` after completion of action a the instruction to the left of the current instruction is executed. Otherwise the instruction to its right is executed.
- $-a$ is the *negative test instruction*. This instruction mirrors the behavior of $+a$, in that it transfers control to the left or right if the action a yields `false` or `true`, respectively.

These instructions are syntactically indistinguishable from PGA's test instructions, but they differ semantically. We define a code semigroup $C' = (\mathcal{I}^+_{C'}, \_;\_)$ with

$$\mathcal{I}_{C'} = (\mathcal{I}_C - \mathfrak{P} - \mathfrak{N}) \cup \bigcup_{a \in \mathcal{A}} \{+a, -a\}.$$

C''s semantics can be formalized by altering the set of equations (3.2): the cases related to $\sigma_i(X) \in \{+/a, -/a, +\backslash a, -\backslash a\}$ are no longer applicable, while two cases to handle $\sigma_i(X) \in \{+a, -a\}$ need to be added. Thus we define for all $i \in \mathbb{Z}$ and $X \in \mathcal{I}^+_{C'}$,

$$|X|^i_{C'} = \begin{cases} \mathsf{D} & \text{if } i < 1 \text{ or } i > \ell(X), \\ a \circ |X|^{i+1}_{C'} & \text{if } \sigma_i(X) = /a, \\ a \circ |X|^{i-1}_{C'} & \text{if } \sigma_i(X) = \backslash a, \\ |X|^{i-1}_{C'} \trianglelefteq a \trianglerighteq |X|^{i+1}_{C'} & \text{if } \sigma_i(X) = +a, \\ |X|^{i+1}_{C'} \trianglelefteq a \trianglerighteq |X|^{i-1}_{C'} & \text{if } \sigma_i(X) = -a, \\ |X|^{i+k}_{C'} & \text{if } \sigma_i(X) = /\#k, \\ |X|^{i-k}_{C'} & \text{if } \sigma_i(X) = \backslash\#k, \\ \mathsf{D} & \text{if } \sigma_i(X) = \#, \\ \mathsf{S} & \text{if } \sigma_i(X) = !. \end{cases}$$

3.4.1 Behavior Preserving Homomorphisms

Now that the behavior of every C'-expression has been specified, we can answer the question whether C' is more or less expressive than C. It turns out that these code semigroups are equally expressive, because we can define behavior preserving homomorphisms from C to C' and vice versa.

First, we define a homomorphism $f\colon \mathcal{I}^+_C \to \mathcal{I}^+_{C'}$ on individual instructions as follows:

$$\begin{array}{llll} /a \mapsto /a;/\#4;\#;\#;\backslash\#4, & +/a \mapsto /\#2;/\#4;+a;/\#7;\backslash\#2, & \# \mapsto \#;\#;\#;\#;\#, \\ \backslash a \mapsto /\#4;\#;\#;\backslash\#4;\backslash a, & -/a \mapsto /\#2;/\#4;-a;/\#7;\backslash\#2, & ! \mapsto !;\#;\#;\#;!, \\ /\#k \mapsto /\#5k;\#;\#;\#;\backslash\#4, & +\backslash a \mapsto /\#2;\backslash\#2;+a;\backslash\#9;\backslash\#2, & \\ \backslash\#k \mapsto /\#4;\#;\#;\#;\backslash\#5k, & -\backslash a \mapsto /\#2;\backslash\#2;-a;\backslash\#9;\backslash\#2. & \end{array}$$

Every C instruction is mapped onto five C' instructions. Observe that f is left-right uniformly behavior preserving. An alternative definition of f could map every C instruction

onto *four* C' instructions, at the expense of being only left *or* right uniformly behavior preserving.

The same holds for the homomorphism $g\colon \mathcal{I}^+_{C'} \to \mathcal{I}^+_C$. One can define left or right uniformly behavior preserving homomorphisms which map every C' instruction onto three C instructions. Here, however, we define g such that it is left-right uniformly behavior preserving:

$$
\begin{aligned}
/a &\mapsto /a; /\#3; \#; \backslash\#3, & /\#k &\mapsto /\#4k; \#; \#; \backslash\#3,\\
\backslash a &\mapsto /\#3; \#; \backslash\#3; \backslash a, & \backslash\#k &\mapsto /\#3; \#; \#; \backslash\#4k,\\
+a &\mapsto +/a; \backslash\#2; /\#2; \backslash\#3, & \# &\mapsto \#; \#; \#; \#,\\
-a &\mapsto -/a; \backslash\#2; /\#2; \backslash\#3, & ! &\mapsto !; \#; \#; !.
\end{aligned}
$$

Chapter 4

Cg Instruction Sequences

The semigroup C introduced in the previous chapter provides two ways to skip one or more instructions during execution: using a test instruction and using a jump instruction. In both cases the location of the target instruction (if present) is at a fixed distance from the source instruction. In other words, the distance over which control of execution is transferred is static and does not depend on the context (i.e., the instructions surrounding the instruction which is currently being executed). As a result, inserting a single instruction at an arbitrary position in some instruction sequence may completely alter its semantics.

To alleviate this problem somewhat, we will introduce an alternative means to transfer control of execution over arbitrary distances within an instruction sequence. This chapter defines the semigroup Cg, a close cousin of C. Cg employs *label instructions* to mark specific positions within an instruction sequence with a natural number (a *label number*). *Goto instructions* can then specify such a label number as the target of a jump.

Cg's instruction set is introduced in §4.1. The semantics of Cg-expressions are formalized in §4.2. This chapter then proceeds with §4.3, §4.4 and §4.5 in which certain properties of label and goto instructions are analyzed and in which some useful transformations of Cg-expressions are defined. Combined, these sections provide us with the tools required to analyze Cg and its relation to C and PGA in Chapter 5. Finally, §4.6 briefly discusses an alternative semantics for goto instructions. After defining behavior preserving endomorphisms on Cg to demonstrate that this alternative semantics does not affect Cg's expressiveness, we will not consider it any further.

4.1 The Instruction Set

The semigroup Cg has basic instructions as well as positive and negative test instructions, just like C. Cg does not have relative jumps $/\#k$ and $\backslash\#k$, unlike C. Instead, it has a set of label instructions $\mathfrak{L}$ and a set of goto instructions $\mathfrak{G}$:[1]

$$\mathfrak{L} = \bigcup_{l\in\mathbb{N}} \{/£l, \backslash £l\}, \qquad \mathfrak{G} = \bigcup_{l\in\mathbb{N}} \{/\#\#£l, \backslash\#\#£l\}.$$

Label instructions mark a specific location within an instruction sequence with a natural number l. They come in a forward as well as a backward oriented flavor, which determines whether the instruction to respectively the right or left of the label instruction is executed next. Goto instructions too are marked with a natural number l and jump to the first label l with the same orientation in the appropriate direction.

[1] The notation for label and goto instructions is borrowed from [BL02, PvdZ06], which define a language PGLDg as part of the PGA language hierarchy. In PGLDg, there are label instructions $£l$ and goto instructions $\#\#£l$, for all $l \in \mathbb{N}$.

Formally, the instruction set $\mathcal{I}_{Cg} = \mathfrak{L} \cup \mathfrak{G} \cup \mathcal{I}_C - \mathfrak{J}$ generates the semigroup $Cg = (\mathcal{I}^+_{Cg}, \_;\_)$. Note that since Cg has basic instructions and test instructions, Cg takes an implicit parameter $\mathcal{A}$ of actions, just like C. Examples of Cg-expressions include:

$$+/a;\#, \qquad /b;/\#\#£0;/a;/£0;!, \qquad /b;/£3;+/a;\backslash\#\#£3, \qquad \backslash£5;-\backslash c. \tag{4.1}$$

Before formalizing Cg's semantics, let us first informally describe what the intended behavior of labels and gotos is.

$/£l$ is a *forward label instruction*. Execution of $/£l$ simply causes the instruction to its right to be executed, if it exists. Otherwise deadlock occurs.

$\backslash£l$ is a *backward label instruction*. It is analogous to $/£l$, except that execution continues with the instruction to its left.

$/\#\#£l$ is a *forward goto instruction*. Transfers control of execution to the nearest $/£l$ instruction to its right, if such an instruction exists. Otherwise deadlock occurs.

$\backslash\#\#£l$ is a *backward goto instruction*. This instruction will cause execution to continue at the nearest $\backslash£l$ instruction to its left. And of course, if such a label does not exist, deadlock will result.

For convenience we will write $\mathfrak{LG}$ for the set $\mathfrak{L} \cup \mathfrak{G}$. The function $\lambda\colon \mathfrak{LG} \to \mathbb{N}$ returns the label number of a given label or goto instruction (e.g., $\lambda(/£6) = 6$). As with C-instructions, we will define two sets $\mathcal{I}^{\rightarrow}_{Cg} \subset \mathcal{I}_{Cg}$ and $\mathcal{I}^{\leftarrow}_{Cg} \subset \mathcal{I}_{Cg}$, which consist of forward and backward Cg-instructions respectively. That is,

$$\mathcal{I}^{\rightarrow}_{Cg} = (\mathcal{I}^{\rightarrow}_{C} \cap \mathcal{I}_{Cg}) \cup \bigcup_{l \in \mathbb{N}} \{/£l, /\#\#£l\},$$
$$\mathcal{I}^{\leftarrow}_{Cg} = (\mathcal{I}^{\leftarrow}_{C} \cap \mathcal{I}_{Cg}) \cup \bigcup_{l \in \mathbb{N}} \{\backslash£l, \backslash\#\#£l\}.$$

Clearly $\mathcal{I}^{\rightarrow}_{Cg} \cap \mathcal{I}^{\leftarrow}_{Cg} = \emptyset$ and $\mathcal{I}^{\rightarrow}_{Cg} \cup \mathcal{I}^{\leftarrow}_{Cg} \cup \{\#, !\} = \mathcal{I}_{Cg}$. The sets $\mathfrak{L}^{\rightarrow}$, $\mathfrak{L}^{\leftarrow}$, $\mathfrak{G}^{\rightarrow}$, $\mathfrak{G}^{\leftarrow}$, $\mathfrak{LG}^{\rightarrow}$ and $\mathfrak{LG}^{\leftarrow}$ are defined as one would expect them to be. Likewise for the directionality relation $\sim \subset \mathcal{I}_{Cg} \times \mathcal{I}_{Cg}$.

Examples We will formalize Cg's semantics in §4.2 below. Still, to create or improve an intuitive understanding of Cg-expressions and how they differ from C-expressions, let us briefly describe the behavior of the Cg-inseqs of (4.1). As before, we specify that execution starts at the leftmost position.

- $+/a;\#$: Performs action a, after which deadlock occurs. This Cg-expression is also a valid C-expression.
- $/b;/\#\#£0;/a;/£0;!$: Performs action b and then terminates. Action a is not performed, since the second instruction is a goto instruction which causes execution to continue at position 4.
- $/b;/£3;+/a;\backslash\#\#£3$: Performs action b followed by action a. Then deadlock results. The action a is *not* repeated, regardless of the value returned by the execution environment, because the backward goto instruction will not transfer control of execution to the forward label instruction: their directionality does not match.
- $\backslash£5;-\backslash c$: Deadlock. After execution of a backward label instruction the instruction to its left is executed. Here, no such instruction is present.

Orphaned Goto Instructions A goto instruction in some Cg-inseq X which causes deadlock (by lack of a "matching" label instruction) will be called *orphaned*. In other words, given some $X \in \mathcal{I}_{Cg}^{+}$ and $i \in \mathfrak{G}(X)$, the ith instruction of X is orphaned iff i is an exit position in X.

Note that although some Cg-expression X may contain labels $/\pounds l$ and $\backslash\pounds l$, this does not preclude the possibility that X contains a goto instruction $/\#\#\pounds l$ or $\backslash\#\#\pounds l$ which matches neither of these labels (and is thus orphaned). For example, in the following expression both goto instructions are orphaned:

$$/\pounds 0; \backslash\#\#\pounds 0; /\#\#\pounds 0; \backslash\pounds 0.$$

The C programming language [ISO99, KR88] (not to be confused with the code semigroup C) allows statements within functions to be marked using labels. The statement **goto** lbl; causes program execution to continue at the statement marked with label lbl, provided that lbl is a label *within the same function*. The Java programming language [GJSB05] allows the labeling of code blocks. The statement **break** lbl; is valid *only* inside a block labeled lbl, and indicates that program execution must be resumed after block lbl.[2]

This shows that C and Java, just like the semigroup Cg, restrict the scope of label and **goto** statements. The statements **goto** lbl; and **break** lbl; may prevent successful compilation of a C or Java program X, even when X contains (multiple) statements labeled with lbl, because of non-overlapping scopes.

When a C or Java compiler encounters a **goto** or **break** statement which references a non-existent or out-of-scope label it may[3] yield an error claiming that a certain *label* is *undefined*. Such an error message seems to lay the "blame" for the failure to compile on the non-existence of some label l, rather than on the incorrectly defined **goto** (**break**) statement. Using the term "orphaned" allows us to indicate that some goto instruction does not have a matching label instruction without blaming any specific label instruction or label number.

4.2 Semantics

As goto instructions transfer control to the nearest label instruction (if present) in the appropriate direction, their semantics depend on the position of said label instruction. In order to make this relation precise, we define two *search functions*,

$$\overrightarrow{\text{SEARCH}}(X, i, S) = \min(\{j \mid j \geq i, \sigma_j(X) \in S\} \cup \{\ell(X) + 1\}),$$
$$\overleftarrow{\text{SEARCH}}(X, i, S) = \max(\{j \mid j \leq i, \sigma_j(X) \in S\} \cup \{0\}).$$

$\overrightarrow{\text{SEARCH}}$ performs a forward search in a given inseq X, starting at position i, for any instruction in S. The first position in X containing one such instruction is returned. If no instruction from S is found then the first position outside of X, (i.e., $\ell(X) + 1$) is returned. $\overleftarrow{\text{SEARCH}}$ behaves nearly identical, except that it searches from right to left, and returns 0 if no instruction is found. Both functions have type $\mathcal{I}_{Cg}^{+} \times \mathbb{Z} \times \mathcal{P}(\mathcal{I}_{Cg}) \to \mathbb{N}$, where $\mathcal{P}(\mathcal{I}_{Cg})$ denotes the powerset of $\mathcal{I}_{Cg}$.

As with PGA and C, we will formally define the semantics of Cg-expressions using basic thread algebra. Let $|\_|_{Cg}^{\rightarrow}: \mathcal{I}_{Cg}^{+} \to \text{BTA}^{\text{reg}}$ be the function that yields the behavior of a given Cg-expression when executed starting with the leftmost instruction. That is, $|\_|_{Cg}^{\rightarrow}$ defines its left behavior. Likewise $|\_|_{Cg}^{\leftarrow}: \mathcal{I}_{Cg}^{+} \to \text{BTA}^{\text{reg}}$ yields the right behavior of a given Cg-expression. As with C, we identify $|X|_{Cg}^{\rightarrow}$ and $|X|_{Cg}^{\leftarrow}$ with $|X|_{Cg}^{1}$ and $|X|_{Cg}^{\ell(X)}$, respectively,

[2] It is actually not quite as simple as this, because of Java's support for exception handling. Furthermore, the **continue** keyword can also be supplied with an optional label, but only if said label precedes an iteration statement, not just any code block. Also note that Java (currently) does not provide a "regular" **goto** statement, although the language does identify **goto** as a reserved keyword.

[3] Tested with *gcc 4.3.3* and *javac 1.6.0_14*.

and define auxiliary functions $|\_|^i_{Cg}\colon \mathcal{I}^+_{Cg} \to \mathrm{BTA}^{\mathrm{reg}}$ for all $i \in \mathbb{Z}$, such that for all $X \in \mathcal{I}^+_{Cg}$,

$$|X|^i_{Cg} = \begin{cases} \mathsf{D} & \text{if } i < 1 \text{ or } i > \ell(X), \\ a \circ |X|^{i+1}_{Cg} & \text{if } \sigma_i(X) = /a, \\ |X|^{i+1}_{Cg} \trianglelefteq a \trianglerighteq |X|^{i+2}_{Cg} & \text{if } \sigma_i(X) = +/a, \\ |X|^{i+2}_{Cg} \trianglelefteq a \trianglerighteq |X|^{i+1}_{Cg} & \text{if } \sigma_i(X) = -/a, \\ |X|^{i+1}_{Cg} & \text{if } \sigma_i(X) = /\pounds l, \\ |X|^{\overrightarrow{\mathrm{SEARCH}}(X,i,\{/\pounds l\})}_{Cg} & \text{if } \sigma_i(X) = /\#\#\pounds l, \\ a \circ |X|^{i-1}_{Cg} & \text{if } \sigma_i(X) = \backslash a, \\ |X|^{i-1}_{Cg} \trianglelefteq a \trianglerighteq |X|^{i-2}_{Cg} & \text{if } \sigma_i(X) = +\backslash a, \\ |X|^{i-2}_{Cg} \trianglelefteq a \trianglerighteq |X|^{i-1}_{Cg} & \text{if } \sigma_i(X) = -\backslash a, \\ |X|^{i-1}_{Cg} & \text{if } \sigma_i(X) = \backslash \pounds l, \\ |X|^{\overleftarrow{\mathrm{SEARCH}}(X,i,\{\backslash \pounds l\})}_{Cg} & \text{if } \sigma_i(X) = \backslash\#\#\pounds l, \\ \mathsf{D} & \text{if } \sigma_i(X) = \#, \\ \mathsf{S} & \text{if } \sigma_i(X) = !. \end{cases} \tag{4.2}$$

As with PGA and C, we equate an infinite sequence of left-to-right derivations according to (4.2) which does not yield an action with deadlock:

> If the equations in (4.2) can be applied infinitely often from left to right without ever yielding an action, then the extracted thread is D. (4.3)

This rule is specifically applicable to infinite loops created using label and goto instructions. For example, $|/\pounds 1; \backslash \pounds 2|^1_{Cg} = \mathsf{D}$, because

$$|/\pounds 1; \backslash \pounds 2|^1_{Cg} = |/\pounds 1; \backslash \pounds 2|^2_{Cg} = |/\pounds 1; \backslash \pounds 2|^1_{Cg}.$$

This example allows for an interesting observation: label instructions can act as control structures even in absence of a matching goto instruction. Another example is the program $/\pounds 5; \backslash a$, which left as well as right behavior is described by the equation $P = a \circ P$. In this sense Cg's label instructions are quite unlike labels in C or Java, where labels cannot alter the flow of control in absence of another statement which references said label (such as **goto**).

Cg, like C, characterizes the regular threads (as stated by Proposition 3.1). We will not prove that fact here; instead we refer to Proposition 6.7 in §6.2. For completeness we end this chapter with the left and right behavior of the examples of §4.1:

$$\begin{aligned} |+/a; \#|^{\rightarrow}_{Cg} &= a \circ \mathsf{D}, & |+/a; \#|^{\leftarrow}_{Cg} &= \mathsf{D}, \\ |/b; /\#\#\pounds 0; /a; /\pounds 0; !|^{\rightarrow}_{Cg} &= b \circ \mathsf{S}, & |/b; /\#\#\pounds 0; /a; /\pounds 0; !|^{\leftarrow}_{Cg} &= \mathsf{S}, \\ |/b; /\pounds 3; +/a; \backslash\#\#\pounds 3|^{\rightarrow}_{Cg} &= b \circ a \circ \mathsf{D}, & |/b; /\pounds 3; +/a; \backslash\#\#\pounds 3|^{\leftarrow}_{Cg} &= \mathsf{D}, \\ |\backslash \pounds 5; -\backslash c|^{\rightarrow}_{Cg} &= \mathsf{D}, & |\backslash \pounds 5; -\backslash c|^{\leftarrow}_{Cg} &= c \circ \mathsf{D}. \end{aligned}$$

4.2.1 Accessibility and Exit Positions

The accessibility relation $\rightarrow_X$ defined on C-inseqs X by Definition 3.2 is defined analogously on Cg-expressions. The same holds for the set $\mathcal{R}_{X,i}$ of instruction positions reachable from position i in X and its complement $\overline{\mathcal{R}_{X,i}}$ (see Definition 3.3). The set of exit positions $\mathcal{E}_X$ of a Cg-inseq X is defined as in Definition 3.4.

Note that for Cg-expressions the notion of accessibility and reachability is in a sense more "artificial" than for C-expressions. This is so because for any orphaned goto instruction on

some position i in an inseq X it is the case that either $i \to_X 0$ or $i \to_X \ell(X)+1$, due to the definition of the functions $\overrightarrow{\text{SEARCH}}$ and $\overleftarrow{\text{SEARCH}}$.

We conclude this section with a result analogous to Proposition 3.5.

Proposition 4.1. *Every regular thread can be described by a Cg instruction sequence in which every instruction is reachable from the start instruction.*

Proof. Consider arbitrary $T \in \text{BTA}^{\text{reg}}$, $X \in \mathcal{I}^+_{Cg}$ and $i \in [1, \ell(X)]$ such that $|X|^i_{Cg} = T$. If $\overline{\mathcal{R}_{X,i}} \cap [1, \ell(X)] = \emptyset$, then T, X and i meet the requirements. Otherwise, randomly select some unreachable position $j \in \overline{\mathcal{R}_{X,i}} \cap [1, \ell(X)]$.

To see why j can be removed from X without problems, we need to make two observations. First, any instruction which transfers control of execution to position j must itself be unreachable. Second, any instruction which transfers control of execution over position j must be a goto instruction; the behavior of such instruction will not be affected by the removal of the instruction at position j (for $\sigma_j(X)$ cannot be a matching label instruction).

The result of removing the instruction at position j from X is an inseq X' such that either $|X'|^{i-1}_{Cg} = T$ or $|X'|^i_{Cg} = T$, depending on whether $j < i$ or $j > i$, respectively. This process can be repeated until all unreachable instructions are removed. □

4.3 Normalizing Label Numbers

Cg-expressions can contain identical goto instructions which, when executed, cause a jump to distinct positions within the instruction sequence. Likewise, identical label instructions can occur multiple times within an expression. For example,

$$X = /\#\#£7; /a; /£7; /b; /\#\#£7; /c; /£7. \tag{4.4}$$

Here, even though $1 =_X 5$, it is easy to see that $|X|^1_{Cg} \neq |X|^5_{Cg}$. Informally, we may say that the identical instructions in this expression are not semantically related. In this section we will make the notion of a semantical relation between label and goto instructions more precise. This endeavor is motivated by the observation that reasoning about a Cg-expression X is greatly simplified if any two label and goto instructions in X with the same label number and direction are known to be related in certain ways.

Definition 4.2. Let $X \in \mathcal{I}^+_{Cg}$. If $i, j \in \mathfrak{LG}(X)$, $\sigma_i(X) \sim \sigma_j(X)$ and $\lambda(\sigma_i(X)) = \lambda(\sigma_j(X))$, then the label/goto instructions at positions i and j have the same label number and direction, and are said to *correspond*, written $i \approx_X j$.

If $i \in \mathfrak{G}(X)$, $j \in \mathfrak{L}(X)$ and $i \to_X j$, then the goto instruction at position i *targets* the label instruction at position j, written $i \curvearrowright_X j$.

If $i, j \in \mathfrak{G}(X)$, $i =_X j$ and $\exists k(i \to_X k \wedge j \to_X k)$, then the identical goto instructions at positions i and j are said to be *target equivalent*, written $i \curlyvee_X j$. Note that target equivalent goto instructions can be orphaned. Also, non-target equivalent goto instructions need *not* be distinct, as in (4.4).

Let $\curvearrowright_X^{-1}$ be the inverse of $\curvearrowright_X$. We define

$$\star_X = \curlyvee_X \cup \curvearrowright_X \cup \curvearrowright_X^{-1} \cup \{(i,i) \mid i \in \mathfrak{L}(X)\}.$$

Instructions at positions $i, j \in \mathfrak{LG}(X)$ are *related* iff $i \star_X j$. X is in *label normal form* (*LNF*) iff $i \approx_X j$ implies $i \star_X j$ for all $i, j \in \mathfrak{LG}(X)$. That is, X is in LNF if and only if any pair of corresponding instructions is related.

Proposition 4.3. *For all $X \in \mathcal{I}^+_{Cg}$, $\star_X$ is an equivalence relation on $\mathfrak{LG}(X)$.*

Proof. Let $I = \{(i,i) \mid i \in \mathfrak{L}(X)\}$. $\star_X$ is reflexive since $I \subseteq \star_X$ and $i \curlyvee_X i$ for all $i \in \mathfrak{G}(X)$. $\star_X$ is symmetric because $\curlyvee_X$, $(\curvearrowright_X \cup \curvearrowright_X^{-1})$ and I are. What remains to be proved is that $\star_X$ is transitive. To that end, let i, j and k be distinct program positions with $i \star_X j$ and $j \star_X k$. We distinguish three situations:

- If $i \curlyvee_X j$ then either $j \curlyvee_X k$, in which case $i \curlyvee_X k$, or $j \curvearrowright_X k$, in which case $i \curvearrowright_X k$.
- If $i \curvearrowright_X j$, then $j \curvearrowright_X^{-1} k$, and hence $i \curlyvee_X k$.
- If $i \curvearrowright_X^{-1} j$, then $j \curlyvee_X k$, meaning that $k \curvearrowright_X i$ and hence $i \curvearrowright_X^{-1} k$. (Note that $j \curvearrowright_X k$ will not be the case because that would mean $i = k$, while we defined i and k to be distinct positions.) □

Proposition 4.4. *Let $X \in \mathcal{I}_{Cg}^+$ be in label normal form. Then the following properties hold for all $1 \leq i, j \leq \ell(X)$:*

(a) If $i \in \mathfrak{G}(X)$, $j \in \mathfrak{L}(X)$ and $i \approx_X j$, then $i \curvearrowright_X j$ (label instructions are targeted by every goto instruction with the same label number and directionality).

(b) If $i, j \in \mathfrak{L}(X)$ and $i =_X j$, then $i = j$ (all label instructions in X are distinct).

(c) If $i, j \in \mathfrak{G}(X)$ and $i =_X j$, then $i \curlyvee_X j$ (identical goto instructions are target equivalent).

Proof. Let $X \in \mathcal{I}_{Cg}^+$ be in LNF. Note that $i =_X j$ implies $i \approx_X j$ for all $i, j \in \mathfrak{LG}(X)$. Since X is in LNF, $i \approx_X j$ implies $i \star_X j$. In order, the properties follow from the following identities:

$$\begin{aligned} \star_X \cap (\mathfrak{G}(X) \times \mathfrak{L}(X)) &= \curvearrowright_X, \\ \star_X \cap (\mathfrak{L}(X) \times \mathfrak{L}(X)) &= \{(i,i) \mid i \in \mathfrak{L}(X)\}, \\ \star_X \cap (\mathfrak{G}(X) \times \mathfrak{G}(X)) &= \curlyvee_X. \end{aligned}$$

□

Proposition 4.5. *For any $X \in \mathcal{I}_{Cg}^+$ there exists an $X' \in \mathcal{I}_{Cg}^+$ such that X' is in label normal form and $|X|_{Cg}^i = |X'|_{Cg}^i$ for all $i \in \mathbb{Z}$.*

Proof. Let $X \in \mathcal{I}_{Cg}^+$. $\star_X$ is an equivalence relation on $\mathfrak{LG}(X)$. Let $[i]_{\star_X}$ be the equivalence class of i and let $\mathfrak{LG}(X)/\star_X$ be the quotient set of $\mathfrak{LG}(X)$ by $\star_X$. Let $n = |\mathfrak{LG}(X)/\star_X|$ be the number of equivalence classes. Now select a bijective mapping f from $\mathfrak{LG}(X)/\star_X$ onto $[1, n]$, and construct an inseq X' by changing the label numbers of each label and goto instruction in X such that $\lambda(\sigma_i(X')) = f([i]_{\star_X})$ for all $i \in \mathfrak{LG}(X)$. Then X' is in LNF and clearly $|X|_{Cg}^i = |X'|_{Cg}^i$ for all $i \in \mathbb{Z}$. □

4.4 Freeing Label Numbers

In this section we will briefly describe how certain label numbers can be removed from a Cg-inseq. It turns out that defining certain behavior preserving mappings on Cg instruction sequences is greatly simplified if one can assume that no label or goto instruction in the input inseq has a label number present in some set L.

Definition 4.6. A label number l is *available* in a Cg-expression X if there is no $u \in X$ such that $\lambda(u) = l$. That is, no label or goto instruction in X has label number l. To make a specific label number available, it must be *freed*. For each $l \in \mathbb{N}$ we define an endomorphism F_l which frees label number l in a given Cg-inseq. $\mathrm{F}_l \colon \mathcal{I}_{Cg}^+ \to \mathcal{I}_{Cg}^+$ is defined on individual instructions as follows:

$$\mathrm{F}_l(u) = \begin{cases} /\pounds l'{+}1 & \text{if } u = /\pounds l' \text{ and } l' \geq l. \\ \backslash\pounds l'{+}1 & \text{if } u = \backslash\pounds l' \text{ and } l' \geq l. \\ /\#\#\pounds l'{+}1 & \text{if } u = /\#\#\pounds l' \text{ and } l' \geq l. \\ \backslash\#\#\pounds l'{+}1 & \text{if } u = \backslash\#\#\pounds l' \text{ and } l' \geq l. \\ u & \text{otherwise.} \end{cases} \tag{4.5}$$

Some behavior preserving mappings require several label numbers to be available. Let $L = \langle l_1, l_2, \ldots, l_n \rangle$ be an arbitrary finite sequence of natural numbers. Then F_L is the endomorphism which frees the label numbers in L in order. Formally, $\mathrm{F}_L = \mathrm{F}_{l_n} \circ \cdots \circ \mathrm{F}_{l_2} \circ \mathrm{F}_{l_1}$.

Proposition 4.7. *Let $l \in \mathbb{N}$ and let L be an arbitrary finite sequence of natural numbers. Then the endomorphisms F_l and F_L are left-right uniformly behavior preserving. Moreover, if L is monotonically nondecreasing, then for every $X \in \mathcal{I}^+_{Cg}$, all label numbers in L are available in $\mathrm{F}_L(X)$.*

Proof. F_l maps individual instructions onto individual instructions and alters only the label number of label and goto instructions with a label number $\geq l$. Execution of a label instruction causes the instruction to its left or right to be executed, depending on the label's orientation, but irrespective of the actual label number. It is not hard to see that likewise the position to which goto instructions transfer control of execution is not affected by the application of F_l. Thus F_l is left-right uniformly behavior preserving. As F_L can be decomposed into individual applications of functions $\mathrm{F}_{l_1}, \ldots, \mathrm{F}_{l_n}$, the same holds for F_L.

Since F_l only increments label numbers $\geq l$, any label number $< l$ which is available in some inseq X will also be available in $\mathrm{F}_l(X)$. It follows that if L is monotonically nondecreasing, then all $l \in L$ will be available in $\mathrm{F}_L(X)$. □

4.5 *Cg* and Relative Jumps

Cg does not have explicit relative jump instructions like *C*. Yet in *Cg*, too, some instructions transfer control of execution relative to their own position: basic instructions, test instruction and label instructions do so. For example, the label instruction /£6 transfers control to the instruction to its immediate right, equivalent to a forward relative jump over distance 1.

Section §4.6 below defines endomorphisms on *Cg* in order to simulate an alternative semantics for goto instructions. These endomorphisms map single instructions onto a fixed number b of different instructions. Under those circumstances care must be taken that instructions which perform an implicit relative jump behave properly: all relative jump distances are multiplied by b.

So how does this work? In this section we will describe how relative jumps over distances up to some arbitrary value k can be emulated using label and goto instructions. As a first step, consider the following family of *Cg*-inseqs, defined for every $l \geq 1$ and $k \geq 2$;

$$\mathrm{D}_l^{\rightarrow} = /£l; /\#\#£l{-}1 \qquad \textsc{left}_k = \mathrm{D}_1^{\rightarrow}; \backslash£0; \mathrm{D}_2^{\rightarrow}; \mathrm{D}_3^{\rightarrow}; \ldots; \mathrm{D}_k^{\rightarrow}$$
$$\mathrm{D}_l^{\leftarrow} = \backslash\#\#£l{-}1; \backslash£l \qquad \textsc{right}_k = \mathrm{D}_k^{\leftarrow}; \ldots; \mathrm{D}_3^{\leftarrow}; \mathrm{D}_2^{\leftarrow}; /£0; \mathrm{D}_1^{\leftarrow}$$

The *Cg*-expressions $\textsc{left}_k$ and $\textsc{right}_k$ contain alternating label and goto instructions, and an extra label with label number 0. $\textsc{left}_k$ and $\textsc{right}_k$ are meant to be used as subsequences of larger instruction sequences. Without going into the use of $\backslash£0$ and $/£0$ for now, observe that $\textsc{left}_k$ contains forward label instructions with label numbers 1 though k, each followed by a forward goto instruction with a label number one less than the number of the preceding label instruction. The same holds for $\textsc{right}_k$, except that it contains backward label and goto instructions.

Next, for all $k \in \mathbb{N}$, consider the family of functions $\phi_k \colon \mathcal{I}_{Cg} \to \mathcal{I}_{Cg}$, defined as

$$\phi_k \colon u \mapsto \begin{cases} /\#\#£1 & \text{if } u = /£l \text{ and } l \leq k, \\ \backslash\#\#£1 & \text{if } u = \backslash£l \text{ and } l \leq k, \\ u & \text{otherwise.} \end{cases}$$

The functions ϕ_k map all label instructions with a label number not greater than k to goto instructions with label number 1.

We now combine $\textsc{left}_k$, $\textsc{right}_k$ and ϕ_k to create endomorphisms $\textsc{rel}_k \colon \mathcal{I}^+_{Cg} \to \mathcal{I}^+_{Cg}$, for all $k \geq 2$, defined on individual instructions $u \in \mathcal{I}_{Cg}$ such that,

$$\textsc{rel}_k \colon u \mapsto \begin{cases} \textsc{left}_k; \backslash\#\#£2; \backslash\#\#£1; \phi_k(u); \backslash£0; \textsc{right}_k & \text{if } u \in \mathcal{I}^{\leftarrow}_{Cg}, \\ \textsc{left}_k; /£0; \phi_k(u); /\#\#£1; /\#\#£2; \textsc{right}_k & \text{otherwise.} \end{cases} \tag{4.6}$$

The functions $\textsc{rel}_k$ are not quite left or right behavior preserving. Instead, at some higher level they redefine the semantics of goto instructions with a label number $l \leq k$, such that their behavior mimics that of a relative jump over distance l. As a special case, $/\#\#£0$ and $\backslash\#\#£0$ signify a jump over distance zero and as such yield deadlock.[4] This alternative semantics can be made explicit by defining thread extraction operators $|X|^i_{Cg,k}$ which are analogous to $|X|^i_{Cg}$, except for the fact that the operators $|X|^i_{Cg,k}$ are defined differently for instances where $i \in \{j \in \mathfrak{G}(X) \mid \lambda(\sigma_j(X)) \leq k\}$:

$$|X|^i_{Cg,k} = \begin{cases} \mathsf{D} & \text{if } i < 1 \text{ or } i > \ell(X), \\ a \circ |X|^{i+1}_{Cg,k} & \text{if } \sigma_i(X) = /a, \\ |X|^{i+1}_{Cg,k} \trianglelefteq a \trianglerighteq |X|^{i+2}_{Cg,k} & \text{if } \sigma_i(X) = +/a, \\ |X|^{i+2}_{Cg,k} \trianglelefteq a \trianglerighteq |X|^{i+1}_{Cg,k} & \text{if } \sigma_i(X) = -/a, \\ |X|^{i+1}_{Cg,k} & \text{if } \sigma_i(X) = /£l, \\ \mathsf{D} & \text{if } \sigma_i(X) = /\#\#£0, \\ |X|^{i+l}_{Cg,k} & \text{if } \sigma_i(X) = /\#\#£l \text{ and } 1 \leq l \leq k, \\ |X|^{\overrightarrow{\textsc{search}}(X,i,\{/£l\})}_{Cg,k} & \text{if } \sigma_i(X) = /\#\#£l \text{ and } l > k, \\ a \circ |X|^{i-1}_{Cg,k} & \text{if } \sigma_i(X) = \backslash a, \\ |X|^{i-1}_{Cg,k} \trianglelefteq a \trianglerighteq |X|^{i-2}_{Cg,k} & \text{if } \sigma_i(X) = +\backslash a, \\ |X|^{i-2}_{Cg,k} \trianglelefteq a \trianglerighteq |X|^{i-1}_{Cg,k} & \text{if } \sigma_i(X) = -\backslash a, \\ |X|^{i-1}_{Cg,k} & \text{if } \sigma_i(X) = \backslash £l, \\ \mathsf{D} & \text{if } \sigma_i(X) = \backslash\#\#£0, \\ |X|^{i-l}_{Cg,k} & \text{if } \sigma_i(X) = \backslash\#\#£l \text{ and } 1 \leq l \leq k, \\ |X|^{\overleftarrow{\textsc{search}}(X,i,\{\backslash £l\})}_{Cg,k} & \text{if } \sigma_i(X) = \backslash\#\#£l \text{ and } l > k, \\ \mathsf{D} & \text{if } \sigma_i(X) = \#, \\ \mathsf{S} & \text{if } \sigma_i(X) = !. \end{cases} \tag{4.7}$$

As an example, consider the Cg-inseq $X = /\#\#£3; /£3; /a; /b$ and suppose that we want to interpret all goto instructions with a label number ≤ 7 as relative jumps. Then,

$$|X|^1_{Cg,7} = |X|^4_{Cg,7} = b \circ |X|^5_{Cg,7} = b \circ \mathsf{D}.$$

Observe that the goto instruction on position 1 transfers control of execution to position 4; the label instruction with the matching label number at position 2 is bypassed.

Fixing some $k \geq 2$, observe that $\textsc{rel}_k$ maps every Cg-instruction on $b_k = 4k + 6$ Cg-instructions. $\textsc{rel}_k$ is defined such that the following equality holds:

$$|X|^i_{Cg,k} = |\textsc{rel}_k(X)|^{b_k(i-1)+1}_{Cg} = |\textsc{rel}_k(X)|^{b_k i}_{Cg}.$$

Specifically,

$$|X|^{\rightarrow}_{Cg,k} = |X|^1_{Cg,k} = |\textsc{rel}_k(X)|^1_{Cg} = |\textsc{rel}_k(X)|^{\rightarrow}_{Cg},$$
$$|X|^{\leftarrow}_{Cg,k} = |X|^{\ell(X)}_{Cg,k} = |\textsc{rel}_k(X)|^{\ell(\textsc{rel}_k(X))}_{Cg} = |\textsc{rel}_k(X)|^{\leftarrow}_{Cg}.$$

It follows that the alternative semantics for Cg as defined by (4.7) can be simulated using $\textsc{rel}_k$ and Cg's default thread extraction operator.

[4] See also §3.1.1.

4.6 Label Instructions as More General Jump Targets

Cg's goto instructions are defined such that they transfer control to a label instruction with the same label number *and directionality* in the appropriate direction (if present). An obvious alternative behavior is for goto instructions to jump to a label instruction with the same label number in the appropriate direction, *irrespective of its directionality* (again, provided such instruction is present). Put more informally: instead of "accepting" jumps from a single direction, we may alter Cg's semantics such that label instructions accept jumps originating from goto instructions in either direction. In this section we play with this idea; it turns out that with respect to expressiveness nothing is gained or lost by using such an alternative semantics. Therefore we will not consider this idea beyond this section. As a result, readers may choose to skip this section.

This alternative semantics can be described by a thread extraction operator $|\_, \_|_{Cg'}$ which is nearly identical to the operator $|\_, \_|_{Cg}$ as defined by the set of equations (4.2) and rule (4.3), except for the cases involving goto instructions. Specifically (now using the usual shorthand notation $|X|^i_{Cg'}$ instead of $|i, X|_{Cg'}$):

$$|X|^i_{Cg'} = \begin{cases} \mathsf{D} & \text{if } i < 1 \text{ or } i > \ell(X), \\ a \circ |X|^{i+1}_{Cg'} & \text{if } \sigma_i(X) = /a, \\ |X|^{i+1}_{Cg'} \trianglelefteq a \trianglerighteq |X|^{i+2}_{Cg'} & \text{if } \sigma_i(X) = +/a, \\ |X|^{i+2}_{Cg'} \trianglelefteq a \trianglerighteq |X|^{i+1}_{Cg'} & \text{if } \sigma_i(X) = -/a, \\ |X|^{i+1}_{Cg'} & \text{if } \sigma_i(X) = /£l, \\ |X|^{\overrightarrow{\text{SEARCH}}(X,i,\{/£l,\backslash £l\})}_{Cg'} & \text{if } \sigma_i(X) = /\#\#£l, \\ a \circ |X|^{i-1}_{Cg'} & \text{if } \sigma_i(X) = \backslash a, \\ |X|^{i-1}_{Cg'} \trianglelefteq a \trianglerighteq |X|^{i-2}_{Cg'} & \text{if } \sigma_i(X) = +\backslash a, \\ |X|^{i-2}_{Cg'} \trianglelefteq a \trianglerighteq |X|^{i-1}_{Cg'} & \text{if } \sigma_i(X) = -\backslash a, \\ |X|^{i-1}_{Cg'} & \text{if } \sigma_i(X) = \backslash £l, \\ |X|^{\overleftarrow{\text{SEARCH}}(X,i,\{/£l,\backslash £l\})}_{Cg'} & \text{if } \sigma_i(X) = \backslash\#\#£l, \\ \mathsf{D} & \text{if } \sigma_i(X) = \#, \\ \mathsf{S} & \text{if } \sigma_i(X) = !. \end{cases}$$

Observe that $\overrightarrow{\text{SEARCH}}$ and $\overleftarrow{\text{SEARCH}}$ now each search for *two* instructions, namely $/£l$, $\backslash £l$, for some $l \in \mathbb{N}$.

4.6.1 Behavior Preserving Homomorphisms

It turns out that this alternative semantics does not affect Cg's expressiveness. It is straightforward to define a homomorphism f such that $|X|^i_{Cg} = |f(X)|^i_{Cg'}$ for all $i \in \mathbb{Z}$ and $X \in \mathcal{I}^+_{Cg}$. f is defined on individual instructions $u \in \mathcal{I}_{Cg}$ such that,

$$f\colon u \mapsto \begin{cases} /£2l & \text{if } u = /£l, \\ \backslash £2l{+}1 & \text{if } u = \backslash £l, \\ /\#\#£2l & \text{if } u = /\#\#£l, \\ \backslash\#\#£2l{+}1 & \text{if } u = \backslash\#\#£l, \\ u & \text{otherwise.} \end{cases}$$

Indeed f ensures that any label number l is even for forward label and goto instructions, while l is odd for backward oriented instructions. As a result, label instructions in $f(X)$ will in practice "accept" jumps from goto instructions in only one direction, rendering the difference between $|\_, \_|_{Cg}$ and $|\_, \_|_{Cg'}$ irrelevant.

Conversely, there exists a homomorphism g such that for all $i \in \mathbb{Z}$ there exists some $j \in \mathbb{Z}$ such that $|X|^i_{Cg'} = |g(X)|^j_{Cg}$. We define $g = \phi \circ \text{REL}_2 \circ \text{F}_{\langle 0,1,2\rangle}$. The functions $\text{F}_{\langle 0,1,2\rangle}$ and REL_2 have been defined previously by (4.5) and (4.6), respectively. The function ϕ is a homomorphism, defined on individual Cg-instructions u such that,

$$\phi\colon u \mapsto \begin{cases} /\#\#£l; \backslash £l; /£l & \text{if } u = /£l \text{ with } l > 2, \\ \backslash £l; /£l; \backslash\#\#£l & \text{if } u = \backslash £l \text{ with } l > 2, \\ u & \text{otherwise.} \end{cases}$$

The correctness of g hinges on three observations:

1. By Proposition 4.7, $\text{F}_{\langle 0,1,2\rangle}$ is behavior preserving.
2. The homomorphism REL_2 alters the semantics of goto instructions with label numbers ≤ 2. These instructions are not present in its input because it is passed the output of $\text{F}_{\langle 0,1,2\rangle}$. As such, $\text{REL}_2 \circ \text{F}_{\langle 0,1,2\rangle}$ is also behavior preserving.
3. Lastly, ϕ does not replace label instructions introduced by REL_2. It *does* replace all other label instructions, such that the resulting subsequence of three instructions mimics the behavior of label instructions as defined by $|\_, \_|_{Cg'}$ if fed to $|\_, \_|_{Cg}$. Any label replaced by ϕ is embedded by REL_2, ensuring that the behavior of other label, basic and test instructions is unaffected. This explains the use of REL_2: it accommodates for the implicit relative jumps performed by these instructions.

We conclude with the observation that g is left-right behavior preserving, but not uniformly so. This is because the number of instructions output by ϕ depends on its input. g can be made left-right uniformly behavior preserving by using an alternative definition of ϕ which always outputs three instructions:

$$\phi\colon u \mapsto \begin{cases} /\#\#£l; \backslash £l; /£l & \text{if } u = /£l \text{ with } l > 2, \\ \backslash £l; /£l; \backslash\#\#£l & \text{if } u = \backslash £l \text{ with } l > 2, \\ u; /\#\#£1; /\#\#£2 & \text{if } u \in \mathcal{I}^{\rightarrow}_{Cg} \wedge u \neq /£l \text{ for } l > 2, \\ \backslash\#\#£2; \backslash\#\#£1; u & \text{otherwise.} \end{cases}$$

Chapter 5

Translating Instruction Sequences

Previous chapters introduced the program algebra PGA and the code semigroups C and Cg. In this chapter we provide behavior preserving mappings between these algebras and show some properties of these translations.

Though defined on at a syntactic level, a behavior preserving mapping f makes explicit certain ways in which (groups of) instructions are related on a semantic level. If $f\colon A \to B$, then f tells us something about distinctions and similarities between code semigroups A and B. If $f\colon A \to A$, then f (if it is not the identity function), can be seen as a reformulation instead of a translation. Additionally, if f is an (anti-)homomorphism then it provides some additional implicit information about how A and B are related. Specifically, it shows that an A-inseq X can be translated instruction by instruction, independent of context, and without taking the length of X as an explicit parameter, to some B-inseq Y. For this reason we aim to define homomorphic instead of arbitrary translations between code semigroups where possible.[1]

The translations defined in this chapter will aid us in proving some expressiveness results in the next chapter. In order, this chapter provides a translation from C to PGA (§5.1), from PGA to C (§5.2), from C to Cg (§5.3) and from Cg to C (§5.4).

5.1 Translating C to PGA

In this section we define a behavior preserving mapping $\textsc{c2pga}\colon \mathcal{I}_C^+ \to \boldsymbol{P}$. We do so in three steps: the first two steps apply left behavior preserving mappings to C itself, thereby converting every C-inseq X to a behaviorally equivalent C-inseq Y which has certain structural properties. The third step exploits these properties in order to translate every such Y to a behaviorally equivalent PGA term Z. The translation presented here is based on the behavior preserving mapping from C onto PGA as defined in section 12 of [BP09a].

1. PGA has basic instructions and test instructions whose semantics are identical to C's forward basic and test instructions. C's backward basic instructions and test instructions have no direct counterpart in PGA, so we wish to eliminate them. Thus we define a left uniformly behavior preserving endomorphism f on $\mathcal{I}_C^+$ which removes these backward

[1] Thinking of A as a high level programming language and B as a lower level programming language or even machine code, we can view f as an interpreter or compiler. If f is an (anti-)homomorphism then parts of an A-inseq X can be transformed and possibly even executed before all of X has been read.

instructions. f is defined on individual instructions as follows:

$$\begin{aligned} /a &\mapsto /a;/\#2;\#, & \backslash a &\mapsto /a;\backslash\#4;\#,\\ +/a &\mapsto +/a;/\#2;/\#4, & +\backslash a &\mapsto +/a;\backslash\#4;\backslash\#8,\\ -/a &\mapsto -/a;/\#2;/\#4, & -\backslash a &\mapsto -/a;\backslash\#4;\backslash\#8, \end{aligned}$$

$$\begin{aligned} /\#k &\mapsto /\#3k;\#;\#, & \# &\mapsto \#;\#;\#,\\ \backslash\#k &\mapsto \backslash\#3k;\#;\#, & ! &\mapsto !;\#;\#. \end{aligned}$$

2. In [BP09a] the notion of *C-programs* is introduced. In essence, a C-program is a C-inseq which does not contain exit positions. I.e, no instruction transfers control of execution outside of the instruction sequence; only execution of the termination or abort instruction will cause program execution to halt. Every C-inseq X can be converted to a C-program, simply by prefixing and suffixing sufficiently many abort instructions. In order to maintain X's left and right behavior, additional jump instructions must be added to its left and right. Let $m \geq 2$ be an upper bound on the largest jump counter present in some C-inseq X. Then a left-right behaviorally equivalent C-program X' can be constructed as
$$/\#m{+}1;(\#)^m;X;(\#)^m;\backslash\#m{+}1.$$
Let g be the left-right behavior preserving mapping which performs the above procedure for arbitrary C-inseqs.

3. Given f and g as defined in the previous two steps, it is immediate that for every C-inseq X there exists a left behaviorally equivalent C-program $X' = g(f(X))$ which does not contain instructions from the set $\mathfrak{B}^{\leftarrow} \cup \mathfrak{P}^{\leftarrow} \cup \mathfrak{N}^{\leftarrow}$. Let $X' = u_1;\ldots;u_n$. Then the following is a behaviorally equivalent PGA term:
$$(\phi_n(u_1);\ldots;\phi_n(u_n))^{\omega}.$$
For all $n \in \mathbb{N}^+$ the function ϕ_n is defined as follows (observe that due to application of g, necessarily $k < n$ and thus $n - k \in \mathbb{N}^+$):
$$\begin{aligned} /a &\mapsto a, & /\#k &\mapsto \#k, & ! &\mapsto !.\\ +/a &\mapsto +a, & \backslash\#k &\mapsto \#n{-}k,\\ -/a &\mapsto -a, & \# &\mapsto \#0, \end{aligned}$$
Denoting the above procedure by h, we have that $\textsc{c2pga} = h \circ g \circ f$.

5.2 Translating PGA to *C*

Defining a translation $\textsc{pga2c}\colon \boldsymbol{P} \to \mathcal{I}_C^+$ turns out to be be a lot easier if PGA terms can be assumed to be in second canonical form. Hence we start out by defining
$$\textsc{pga2c} = \textsc{snd2c} \circ \textsc{snd}\,.$$
Recall that $\textsc{snd}\colon \boldsymbol{P} \to \boldsymbol{P_2}$ is the function defined in §2.2.2 which converts arbitrary PGA terms to their structurally (and behaviorally) equivalent minimal second canonical forms. The mapping $\textsc{snd2c}\colon \boldsymbol{P_2} \to \mathcal{I}_C^+$ is a behavior preserving mapping defined on second canonical forms only. Any $X \in \boldsymbol{P_2}$ does not contain chained jump instructions and has one of two forms:

- X does not contain repetition and thus $X = u_1;u_2;\ldots;u_n$ for some $n \in \mathbb{N}^+$. We define
$$\textsc{snd2c}(u_1;u_2;\ldots;u_n) = \psi(u_1);\psi(u_2);\ldots;\psi(u_n).$$

- $X = Y; Z^\omega$, and Y nor Z contain repetition, meaning that for some $n, m \in \mathbb{N}^+$, $X = u_1; \ldots ; u_n; (u_{n+1}; \ldots ; u_{n+m})^\omega$. Now we define

$$\textsc{snd2c}(u_1; \ldots ; u_n; (u_{n+1}; \ldots ; u_{n+m})^\omega) = \psi(u_1); \ldots ; \psi(u_{n+m}); (\backslash\#m)^{\max(2,m-1)}.$$

The function ψ is as straightforward as can be:

$$a \mapsto /a \qquad +a \mapsto +/a \qquad -a \mapsto -/a \qquad ! \mapsto ! \qquad \#l \mapsto \begin{cases} \# & \text{if } l = 0, \\ /\#l & \text{otherwise.} \end{cases}$$

SND2C makes extensive use of the assumptions that can be made about its input (i.e., that it is in second canonical form). Any jump instruction u_i with $i \leq n$ will not jump beyond u_{n+m}. Any jump instruction u_i with $i > n$ will not have a jump counter greater than $m-1$. By appending $\max(2, m-1)$ $\backslash\#m$ instructions, it is ensured that all jump instructions which transfer control of execution beyond $\psi(u_{n+m})$ indirectly transfer control to the appropriate instruction. Since u_{n+m-1} and u_{n+m} can be test instructions, it is important to append at least two backward jump instructions.

5.3 Translating C to Cg

In this section we focus on translations from C to Cg. It turns out that there does not exist a homomorphism which translates arbitrary C-expressions to behaviorally equivalent Cg-expressions. Theorem 5.1 below gives a proof of this fact.

A convenient way to translate C to Cg is to start out by categorizing every C-expression based on the largest jump counter it contains. We write $C_{\leq k}$ for the subsemigroup of C which consists exactly of those C-expressions that do not contain instructions $/\#k'$ or $\backslash\#k'$ for $k' > k$. Formally, $C_{\leq k} = (\mathcal{I}^+_{C_{\leq k}}, \_;\_)$, with[2]

$$\mathcal{I}_{C_{\leq k}} = \mathcal{I}_C - \{u \in \mathfrak{J} \mid \delta(u) > k\}. \tag{5.1}$$

Assume the existence of a family of behavior preserving mappings $\textsc{c2cg}_k \colon \mathcal{I}^+_{C_{\leq k}} \to \mathcal{I}^+_{Cg}$ for all $k \in \mathbb{N}$. Writing $\textsc{c2cg}(k, X)$ for $\textsc{c2cg}_k(X)$, the behavior preserving mapping $\textsc{c2cg} \colon \mathcal{I}^+_C \to \mathcal{I}^+_{Cg}$ can then be defined on all $X \in \mathcal{I}^+_C$ as,[3]

$$\textsc{c2cg} \colon X \mapsto \textsc{c2cg}(\max\{\delta(\sigma_i(X)) \mid i \in \mathfrak{J}(X)\}, X).$$

The hypothesized family of functions $\textsc{c2cg}_k$ exists. A straightforward definition is (5.3) in §5.3.1 below. An alternative homomorphic definition is (5.5) in §5.3.2. Since in both cases $\textsc{c2cg}_k$ is only defined for $k \geq 2$, a slightly altered definition of $\textsc{c2cg} \colon \mathcal{I}^+_C \to \mathcal{I}^+_{Cg}$ is in place:

$$\textsc{c2cg} \colon X \mapsto \textsc{c2cg}(\max(\{\delta(\sigma_i(X)) \mid i \in \mathfrak{J}(X)\} \cup \{2\}), X). \tag{5.2}$$

5.3.1 A Behavior Preserving Mapping from $C_{\leq k}$ to Cg

For all $k \geq 2$, we define a function $\textsc{c2cg}_k \colon \mathcal{I}^+_{C_{\leq k}} \to \mathcal{I}^+_{Cg}$ such that,

$$\textsc{c2cg}_k(u_1; \ldots ; u_n) = \psi_{k,1}(u_1); \ldots ; \psi_{k,n}(u_n). \tag{5.3}$$

[2] Recall that $\delta \colon \mathfrak{J} \to \mathbb{N}^+$ returns the jump counter of a given jump instruction.

[3] Yes, the function name C2CG is overloaded here. Its type is either $\mathbb{N} \times \mathcal{I}^+_C \to \mathcal{I}^+_{Cg}$ or simply $\mathcal{I}^+_C \to \mathcal{I}^+_{Cg}$.

In effect $\textsc{c2cg}_k$ replaces the ith instruction of its input X with the output of $\psi_{k,i}(\sigma_i(X))$. The auxiliary functions $\psi_{k,i}\colon \mathcal{I}_C \to \mathcal{I}_C^+$ are defined as follows:

$$\psi_{k,i}(u) = \begin{cases} \phi_{k,i}(/a; /\#\#£[i{+}1]_{k+1}) & \text{if } u = /a, \\ \phi_{k,i}(+/a; /\#\#£[i{+}1]_{k+1}; /\#\#£[i{+}2]_{k+1}) & \text{if } u = +/a, \\ \phi_{k,i}(-/a; /\#\#£[i{+}1]_{k+1}; /\#\#£[i{+}2]_{k+1}) & \text{if } u = -/a, \\ \phi_{k,i}(/a; \backslash\#\#£[i{-}1]_{k+1}) & \text{if } u = \backslash a, \\ \phi_{k,i}(+/a; \backslash\#\#£[i{-}1]_{k+1}; \backslash\#\#£[i{-}2]_{k+1}) & \text{if } u = +\backslash a, \\ \phi_{k,i}(-/a; \backslash\#\#£[i{-}1]_{k+1}; \backslash\#\#£[i{-}2]_{k+1}) & \text{if } u = -\backslash a, \\ \phi_{k,i}(/\#\#£[i{+}l]_{k+1}) & \text{if } u = /\#l, \\ \phi_{k,i}(\backslash\#\#£[i{-}l]_{k+1}) & \text{if } u = \backslash\#l, \\ \phi_{k,i}(\#) & \text{if } u = \#. \\ \phi_{k,i}(!) & \text{if } u = !. \end{cases}$$

In this definition $[n]_{k+1}$ stands for the remainder of n after division by $k{+}1$, i.e. the smallest nonnegative value congruent with $n \pmod{k+1}$. Thus $0 \leq [n]_{k+1} \leq k$ for all n. For all $i \in \mathbb{N}^+$, $\phi_{k,i}\colon \mathcal{I}_{Cg}^+ \to \mathcal{I}_{Cg}^+$ embeds its argument between some label and goto instructions with label number $[i]_{k+1}$ as follows:

$$\phi_{k,i}(U) = /\#\#£[i]_{k+1}; \backslash£[i]_{k+1}; /£[i]_{k+1}; U; \backslash\#\#£[i]_{k+1}.$$

Informally, $\phi_{k,i}(U)$ "guards" the Cg instruction sequence U which replace the C instruction at position i in the original C-expression using the labels $/£[i]_{k+1}$ and $\backslash£[i]_{k+1}$. In this way a goto instruction $/\#\#£[i+l]_{k+1}$ or $\backslash\#\#£[i+l]_{k+1}$ in a Cg-inseq U which replaces the ith C instruction transfers execution to the Cg-inseq U' which replaces the C instruction at position $i+l$ or $i-l$, respectively. In this way the transfer of control of execution over a relative distance in the original C-inseq is simulated.

Observe that label numbers are repeated ("reused") with period $k+1$. This does not pose a problem because the original $C_{\leq k}$-expression will not contain relative jumps over a distance greater than k. (And since $k \geq 2$, the implicit relative jumps over distance 1 or 2 performed by test instructions can likewise be simulated.)

The auxiliary functions $\psi_{k,i}$ and their helper functions $\phi_{k,i}$ are defined such that $\textsc{c2cg}_k$ is left-right behavior preserving. Note that it is possible to omit the rightmost $\backslash\#\#£[i]_{k+1}$ instruction outputted by each call to $\phi_{k,i}$, but then $\textsc{c2cg}_k$ would no longer be right behavior preserving.

5.3.2 What About a Homomorphic Translation from C to Cg?

The translation $\textsc{c2cg}\colon \mathcal{I}_C^+ \to \mathcal{I}_{Cg}^+$ defined by (5.2) is not homomorphic because it requires knowledge about the largest jump counter present in its input. It turns out that it is not possible to define a homomorphic alternative to C2CG.

Theorem 5.1. *There does not exist a behavior preserving homomorphism $f\colon \mathcal{I}_C^+ \to \mathcal{I}_{Cg}^+$.*

Proof. We prove that no homomorphism $f\colon \mathcal{I}_C^+ \to \mathcal{I}_{Cg}^+$ can be left behavior preserving. The proof that no such f can be right behavior preserving is analogous.

For all $n \in \mathbb{N}^+$ we define the following C-inseqs:

$$\begin{aligned} \textsc{node}_n &= +/a; /\#3n{-}1; /\#3n{+}1, \\ \textsc{tree}_n &= \textsc{node}_1; \textsc{node}_2; \ldots; \textsc{node}_{2^n-1}. \end{aligned} \tag{5.4}$$

Observe that $\textsc{tree}_n$ contains 2^n exit positions (see Definition 3.4), each containing one of the rightmost 2^n forward jump instructions of $\textsc{tree}_n$. Exactly one of these exit positions

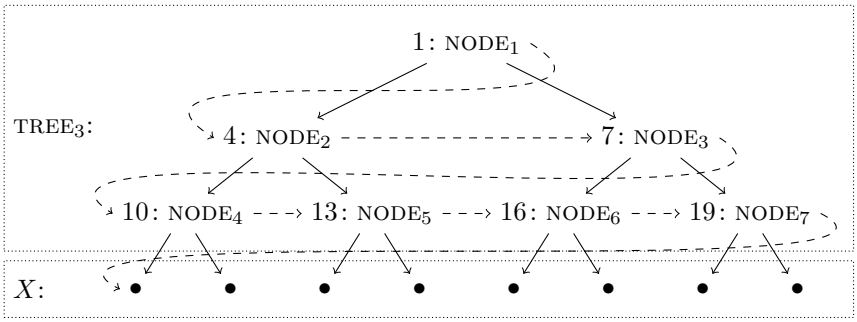

Figure 5.1: Graphical representation of the C-expression $\text{TREE}_3; X$ as defined in (5.4). The dashed arrows show the order in which the subexpressions $\text{NODE}_1, \ldots, \text{NODE}_7$ are concatenated (the prefixes denote their positions in TREE_3). The solid arrows signify jumps between which a choice is made based on the boolean reply to the a-test in the originating node. As depicted here, all instructions at exit positions of TREE_3 jump to distinct positions within the inseq X. This means that $\ell(X) \geq 22$.

will be reached after n consecutive a-tests, provided that execution starts at position 1. Every instruction in TREE_n is reachable from position 1. Figure 5.1 provides a graphical representation of TREE_3.

Towards a contradiction we will now assume that there *does* exist a left behavior preserving homomorphism f from the code semigroup C onto the code semigroup Cg.

It is easy to see that for any combination of $m \leq 2^n$ exit positions $i_1, i_2, \ldots, i_m$ in TREE_n there exist some $X \in \mathcal{I}_C^+$ such that all of the following yield distinct behavior:[4]

$$|\text{TREE}_n; X|_C^{i_1}, |\text{TREE}_n; X|_C^{i_2}, \ldots, |\text{TREE}_n; X|_C^{i_m}.$$

It follows that $f(\text{TREE}_n)$ must have at least $2^n - 2$ distinct orphaned forward goto instructions, all of which are reachable from the leftmost instruction.[5]

For all $X \in \mathcal{I}_{Cg}^+$, let $L_X = X \cap \mathfrak{L}^{\rightarrow}$ be the set of distinct forward label instructions in X. Obviously $L_X = L_{X^k}$ for all $k \in \mathbb{N}^+$.

Now take some $n, k \in \mathbb{N}$ such that $2^n - 2 > |L_{f(/a)}|$ and $k \geq 3(2^n - 1) + 1$. Then $|\text{TREE}_n; (/a)^k|_C^1$ will perform at least $n+1$ consecutive a-actions, irrespective of the boolean replies they yield. However, this cannot be the case for $|f(\text{TREE}_n; (/a)^k)|_{Cg}^1$. Some of the forward goto instructions in $f(\text{TREE}_n)$ which are reachable after n a-tests cannot have a matching label instruction in $f((/a)^k)$, because the number of distinct forward label instructions $|L_{f((/a)^k)}| = |L_{(f(/a))^k}| = |L_{f(/a)}|$ is smaller than the number of distinct forward goto instructions (which is at least $2^n - 2$). Thus we reach a contradiction. □

A Behavior Preserving Homomorphism from $C_{\leq k}$ to Cg It turns out that the result of Theorem 5.1 is due to a surprisingly small lack of information about the context of individual instructions. Once an upper bound on the size of jump counters in the input inseq is known, a homomorphism *can* be defined. In other words, there does exists a homomorphic alternative to the family of behavior preserving mappings $\text{C2CG}_k \colon \mathcal{I}_{C_{\leq k}}^+ \to \mathcal{I}_{Cg}^+$ defined by (5.3) in §5.3.1. We provide one such alternative definition, by building on the work of §4.5. For all $k \geq 2$ we define,

$$\text{C2CG}_k = \text{REL}_k \circ \phi. \tag{5.5}$$

[4] In fact, infinitely many inseqs X have this property.

[5] We do not exclude the possibility that either or both of the rightmost two instruction positions of $f(\text{TREE}_n)$ are exit positions containing forward basic instructions, test instructions or label instructions. This explains the conservative estimate of $2^n - 2$ instead of 2^n orphaned forward goto instructions.

The homomorphism $\textsc{rel}_k\colon \mathcal{I}^+_{Cg} \to \mathcal{I}^+_{Cg}$ is defined by (4.6) in §4.5. Recall that it causes all goto instructions with label numbers up to and including k to behave as relative jumps. It should come as no surprise then that the definition of the homomorphism $\phi\colon \mathcal{I}^+_C \to \mathcal{I}^+_{Cg}$ is straightforward:

$$\phi\colon u \mapsto \begin{cases} /\#\#£k & \text{if } u = /\#k, \\ \backslash\#\#£k & \text{if } u = \backslash\#k, \\ u & \text{otherwise.} \end{cases}$$

Observe that $\textsc{c2cg}_k$ is left-right uniformly behavior preserving. (Like $\textsc{rel}_k$, $\textsc{c2cg}_k$ maps every instruction in the input instruction sequence to $4k + 6$ instructions in the output.)

5.4 Translating *Cg* to *C*

Defining a behavior preserving mapping $\textsc{cg2c}\colon \mathcal{I}^+_{Cg} \to \mathcal{I}^+_C$ is rather straightforward. Label instructions can simply be replaced by relative jumps over distance 1 in the appropriate direction. Goto instructions are replaced by relative jumps to the position of the label instruction which they target, if any. Orphaned goto instructions can be replaced by an abort instruction or a jump outside of the instruction sequence. For convenience we will choose to do the latter.

For all $i \in \mathbb{N}^+$ we define functions $\phi_i\colon \mathcal{I}^{\leqslant i}_{Cg} \to \mathcal{I}_C$ such that,

$$\phi_i(X) = \begin{cases} /\#j{-}i & \text{if } \sigma_i(X) = /\#\#£l \text{ and } j = \overrightarrow{\textsc{search}}(X, i, \{/£l\}), \\ \backslash\#i{-}j & \text{if } \sigma_i(X) = \backslash\#\#£l \text{ and } j = \overleftarrow{\textsc{search}}(X, i, \{\backslash£l\}), \\ /\#1 & \text{if } \sigma_i(X) = /£l, \\ \backslash\#1 & \text{if } \sigma_i(X) = \backslash£l, \\ \sigma_i(X) & \text{otherwise.} \end{cases} \tag{5.6}$$

ϕ_i determines whether and how the ith instruction in a given Cg-inseq X should be translated. Only label and goto instructions are replaced, precisely according to the rules mentioned. Concatenating the results of appropriate invocations of (5.6), the mapping $\textsc{cg2c}\colon \mathcal{I}^+_{Cg} \to \mathcal{I}^+_C$ is thus defined:

$$\textsc{cg2c}\colon X \mapsto \phi_1(X); \phi_2(X); \ldots; \phi_{\ell(X)}(X). \tag{5.7}$$

Every label and goto instruction is replaced by a jump instruction which mimics its transfer of control of execution. Other instructions are unaltered. Thus $\textsc{cg2c}$ is left-right uniformly behavior preserving.

5.4.1 What About a Homomorphic Translation from *Cg* to *C*?

The translation $\textsc{cg2c}$ defined by (5.7) is not a homomorphism. It turns out that this is necessarily so.

Theorem 5.2. *There does not exist a behavior preserving homomorphism* $f\colon \mathcal{I}^+_{Cg} \to \mathcal{I}^+_C$.

Proof. We prove that no homomorphism $f\colon \mathcal{I}^+_{Cg} \to \mathcal{I}^+_C$ can be left behavior preserving. The proof that no such f can be right behavior preserving is analogous.

For all $n \in \mathbb{N}^+$ we define the following Cg-inseqs:

$$\textsc{node}_n = /£n; +/a; /\#\#£2n; /\#\#£2n{+}1,$$
$$\textsc{tree}_n = \textsc{node}_1; \textsc{node}_2; \ldots; \textsc{node}_{2^n-1}.$$

It is not hard to see that TREE_n contains 2^n orphaned goto instructions with label numbers 2^n through $2^{n+1}-1$. For example, TREE_2 contains the orphaned goto instructions $/\#\#£4$, $/\#\#£5$, $/\#\#£6$ and $/\#\#£7$:

$$\begin{aligned}/£1; +/a; /\#\#£2; /\#\#£3;\\ /£2; +/a; /\#\#£4; /\#\#£5;\\ /£3; +/a; /\#\#£6; /\#\#£7.\end{aligned}$$

If execution of TREE_n starts at position 1, then exactly one of the orphaned goto instructions will be reached after performing n consecutive a-actions. Every orphaned goto instruction is reachable.

Towards a contradiction we will now assume that there *does* exist a left behavior preserving homomorphism f from the code semigroup Cg onto the code semigroup C.

For all $X \in \mathcal{I}_C^+$ define $R_X = \{j - \ell(X) \mid i \in [1, \ell(X)], i \to_X^+ j, j > \ell(X)\}$. Informally, R_X contains the offsets of "invalid" positions to the right of X which are reachable from X. We fix some r such that $\max(R_{f(/a)}) \leq \ell(f((/a)^r))$. Then $R_{f(/a;(/a)^r)} = R_{f((/a)^r)}$, and in fact $R_{f((/a)^k)} = R_{f((/a)^r)}$ for all $k \geq r$.

Next we define $\text{TREE}_{n,k} = \text{TREE}_n; (/a)^k$ for all $n, k \in \mathbb{N}$, and we make two easily verifiable claims:

(1) For all $n \in \mathbb{N}$, $k, k' \geq 2$ and $X \in \mathcal{I}_{Cg}^+$ the identity $|\text{TREE}_{n,k}; X|_{Cg}^{\to} = |\text{TREE}_{n,k'}; X|_{Cg}^{\to}$ holds. To see why this is so, observe that all exit positions in $\text{TREE}_{n,k}$ and $\text{TREE}_{n,k'}$ are goto instructions and that $\text{TREE}_{n,k}$ and $\text{TREE}_{n,k'}$ do not contain backward label instructions. As a result only the last two instructions of $\text{TREE}_{n,k}$ and $\text{TREE}_{n,k'}$ (which are $/a$ instructions) may be reachable from a position in the "X-part" of $\text{TREE}_{n,k}; X$ and $\text{TREE}_{n,k'}; X$.

(2) For any combination of $m \leq 2^n$ distinct positions of orphaned goto instructions i_1, $i_2, \ldots, i_m$ within $\text{TREE}_{n,k}$ there exists an $X \in \mathcal{I}_{Cg}^+$ such that all of the following yield distinct behavior:[6]

$$|\text{TREE}_{n,k}; X|_{Cg}^{i_1}, |\text{TREE}_{n,k}; X|_{Cg}^{i_2}, \ldots, |\text{TREE}_{n,k}; X|_{Cg}^{i_m}.$$

Combining these two claims, we must conclude that $|R_{f(\text{TREE}_{n,k})}| \geq 2^n$ for all $n, k \in \mathbb{N}$. Now take some n such that $2^n > |R_{f((/a)^r)}|$ and select some $k \geq r$ such that $R_{f(\text{TREE}_{n,k})} = R_{f(\text{TREE}_n;(/a)^k)} = R_{f((/a)^k)} = R_{f((/a)^r)}$. But then $|R_{f(\text{TREE}_{n,k})}| = |R_{f((/a)^r)}| < 2^n$. Contradiction. □

A Behavior Preserving Homomorphism from $Cg_{\leq k}$ to C Similar to the definition of subsemigroups $C_{\leq k}$, we define subsemigroups $Cg_{\leq k} \subset Cg$ for all $k \in \mathbb{N}$. $Cg_{\leq k}$ contains precisely those Cg-inseqs which do not contain goto instructions with a label number greater than k. That is, we define $Cg_{\leq n} = (\mathcal{I}_{Cg_{\leq n}}^+, \_;\_)$, with

$$\mathcal{I}_{Cg_{\leq k}} = \mathcal{I}_{Cg} - \{u \in \mathfrak{G} \mid \lambda(u) > k\}.$$

Note that $Cg_{\leq k}$ places no restriction on label instructions. As such, the utility of label instructions with a label number greater than k in a $Cg_{\leq k}$-expression is limited.

As per Theorem 5.2 no total homomorphism from Cg to C can be behavior preserving. However, the family of behavior preserving functions $\text{CG2C}_k \colon \mathcal{I}_{Cg_{\leq k}}^+ \to \mathcal{I}_C^+$ ($k \in \mathbb{N}$) can be defined such that each CG2C_k is a homomorphism. Given arbitrary k, we define CG2C_k on individual instructions as follows:

$$\text{CG2C}_k(u) = \phi_k(u); \text{NEXT}_k(u); \text{LEFT}_k(u); \text{RIGHT}_k(u). \tag{5.8}$$

[6]Note again that there are in fact infinitely many such X.

Here ϕ_k is defined as:

$$\begin{array}{cccc} /a \mapsto /a, & \backslash a \mapsto /a, & /£l \mapsto /\#1, & /\#\#£l \mapsto /\#k{+}l{+}4, \\ +/a \mapsto +/a, & +\backslash a \mapsto +/a, & \backslash £l \mapsto /\#1, & \backslash\#\#£l \mapsto /\#l{+}3, \\ -/a \mapsto -/a, & -\backslash a \mapsto -/a, & \# \mapsto \#, & ! \mapsto !. \end{array}$$

Furthermore, NEXT_k, LEFT_k and RIGHT_k are defined as follows:

$$\text{NEXT}_k\colon u \mapsto \begin{cases} \backslash\#2k{+}6; \backslash\#4k{+}12 & \text{if } u \in \mathcal{I}^{\leftarrow}_{Cg}, \\ /\#2k{+}4; /\#4k{+}8 & \text{otherwise,} \end{cases}$$

$$\text{LEFT}_k\colon u \mapsto \begin{cases} (\backslash\#2k{+}5)^l; \backslash\#l{+}3; (\backslash\#2k{+}5)^{k-l} & \text{if } u = \backslash £l \text{ and } l \leq k, \\ (\backslash\#2k{+}5)^{k+1} & \text{otherwise,} \end{cases}$$

$$\text{RIGHT}_k\colon u \mapsto \begin{cases} (/\#2k{+}5)^l; \backslash\#k{+}l{+}4; (/\#2k{+}5)^{k-l} & \text{if } u = /£l \text{ and } l \leq k, \\ (/\#2k{+}5)^{k+1} & \text{otherwise.} \end{cases}$$

The mapping $\text{CG2C}_k\colon X \mapsto Y$ can be explained using the metaphor of a "highway" that is laid between successive instructions of X. The highway contains a dedicated lane for each goto instruction $/\#\#£l$ and $\backslash\#\#£l$ for $0 \leq l \leq k$, thus resulting in a highway with $2k+2$ lanes. The highway is the result of the functions LEFT_k and RIGHT_k. Each $Cg_{\leq k}$-instruction is mapped onto $2k+5$ C-instructions:

$$\underbrace{u; v; w; \overbrace{\backslash\#2k{+}5; \ldots; \backslash\#2k{+}5}^{k+1 \text{ "lanes" to the left}}; \overbrace{/\#2k{+}5; \ldots; /\#2k{+}5}^{k+1 \text{ "lanes" to the right}}}_{\text{these } 2k+5 \ C \text{ instructions represent a single } Cg_{\leq k} \text{ instruction}} \quad \overbrace{\qquad\qquad}^{\text{label/goto "highway" with } 2k+2 \text{ "lanes"}}$$

The highway is used solely to mimic the behavior of goto instructions using a finite number of jumps. The following C-inseq is yielded by $\text{CG2C}_k(/\#\#£l)$:

$$\underbrace{/\#k{+}l{+}4; \overbrace{/\#2k{+}4; /\#4k{+}8;}^{\phi_k(/\#\#£l)} \overbrace{(\backslash\#2k{+}5)^{k+1};}^{\text{LEFT}_k(/\#\#£l)} \overbrace{(/\#2k{+}5)^{l};}^{\text{RIGHT}_k(/\#\#£l)} \overbrace{/\#2k{+}5;}^{\text{right lane } l} (/\#2k{+}5)^{k-l}}_{k+l+3 \text{ instructions}} \quad \overbrace{\qquad\qquad}^{\text{entering the "highway"}}$$

The intention here is that the effect of $/\#\#£l$ is to jump onto the lth highway lane to the right. This lane consists of chained jumps, each of distance $2k+5$, until the segment of C-instructions that is the result of $\text{CG2C}_k(/£l)$ (note that $l \leq k$, for otherwise $/\#\#£l$ would not be part of the input). There, a jump instruction off the highway can be found:

$$/\#1; \underbrace{\overbrace{/\#2k{+}4;}^{\text{to next } Cg_{\leq k} \text{ instruction}} /\#4k{+}8 \overbrace{(\backslash\#2k{+}5)^{k+1};}^{\text{LEFT}_k(/£l)} \overbrace{(/\#2k{+}5)^{l};}^{\text{RIGHT}_k(/£l)} \overbrace{\backslash\#k{+}l{+}4;}^{\text{right lane } l} (/\#2k{+}5)^{k-l}}_{2k+3 \text{ instructions}} \quad \overbrace{\qquad\qquad}^{\text{leaving the "highway"}}$$

CG2C_k maps each $Cg_{\leq k}$-instruction in an inseq X onto $2k+5$ C-instructions in an inseq Y. Thus the C-instructions corresponding to the ith instruction in X start in Y at position $(i-1) \cdot (2k+5) + 1$.

It follows that $|X|^i_{Cg} = |\text{CG2C}_k(X)|^{(i-1)(2k+5)+1}_C$ for all $i \in \mathbb{Z}$, $k \leq 2$ and $X \in \mathcal{I}^+_{Cg}$. Thus CG2C_k is left uniformly behavior preserving.

Chapter 6

Some Expressiveness Results

As stated in §3.1.1, the abort instruction does not enhance C's expressiveness as any abort instruction can be replaced by a jump instruction with a sufficiently large jump counter. In §5.1 the first of three steps involving the translation of C to PGA involved the elimination of backward basic/test instructions. These observations naturally lead one to wonder whether C contains more redundant instructions. There are at least two ways to prove that this is indeed the case, both of which will be utilized in this chapter.

- On the one hand one can define a procedure M which, given an arbitrary regular thread $T \in \mathrm{BTA}^{\mathrm{reg}}$, constructs a C-expression X such that $|X|_C^i = T$ for some $i \in [1, \ell(X)]$, using only a subset of all C instructions, regardless of T. Clearly, any instruction which is not utilized by M irrespective of its input is redundant in the sense that it does not enhance C's expressiveness.
- On the other hand one can define a function f on $\mathcal{I}_C^+$ which translates any given inseq X to a behaviorally equivalent inseq Y, such that certain instructions will never be present in Y. Again, any such instruction can be deemed redundant from the point of view of expressiveness.

In our quest to trim C's instruction set we will inevitably stumble upon instruction sets which cannot express all threads in $\mathrm{BTA}^{\mathrm{reg}}$. As we will later see, there is in fact a hierarchy of expressive power.

Each C or Cg instruction u has a *dual* $\overline{u}$: for forward instructions this is their backward counterpart, and vice versa. The abort and termination instructions are their own dual. Thus e.g. $\overline{/a} = \backslash a$, $\overline{-\backslash b} = -/b$ and $\overline{\#} = \#$. Observe that the dual operator is an involution: $\overline{\overline{u}} = u$ for all $u \in \mathcal{I}_C \cup \mathcal{I}_{Cg}$.

The anti-automorphism REV reverses a given instruction sequence and converts all its instructions to their dual. It is defined on C and well as Cg instruction sequences. For example,

$$\mathrm{REV}(+/a; !; \backslash\#2) = \overline{\backslash\#2}; \overline{!}; \overline{+/a} = /\#2; !; +\backslash a.$$

Observe that REV is an involution, because for all $i \in [1, \ell(X)]$,

$$\sigma_i(X) = \overline{\sigma_{\ell(X)-i+1}(\mathrm{REV}(X))} = \overline{\overline{\sigma_{\ell(X)-(\ell(X)-i+1)+1}(\mathrm{REV} \circ \mathrm{REV}(X))}} = \sigma_i(X).$$

It is not hard to see that $|X|_C^{\rightarrow} = |\mathrm{REV}(X)|_C^{\leftarrow}$ for arbitrary inseq X. It follows that any code semigroup generated by some set $I \subseteq \mathcal{I}_C$ or $I \subseteq \mathcal{I}_{Cg}$ is exactly as expressive as the set of its duals $\{\overline{u} \mid u \in I\}$. Thus REV tells us something about the expressiveness of subsemigroups of C and Cg.

The remainder of this chapter is organized as follows: in §6.1 we will be concerned with the expressiveness of several subsemigroups of C. Specifically, we will show that a reduction of $\mathcal{I}_C$ so that it contains only a finite number of forward or backward jump instructions (or

both) reduces its expressiveness. In §6.2 we will combine the results of §6.1 with some of the translations defined in the previous chapter and use these to make some statements about the expressiveness of Cg and some of its subsemigroups.

6.1 The Expressiveness of Subsemigroups of C

In §5.1 it was shown that backward basic instructions and backward test instructions do not increase C's expressiveness, by means of a left behavior preserving endomorphism f on $\mathcal{I}_C^+$ which does not output any of these instructions. In other words, the code semigroup generated by the instruction set $\mathcal{I}_C - \mathfrak{B}^{\leftarrow} - \mathfrak{P}^{\leftarrow} - \mathfrak{N}^{\leftarrow}$ is as expressive as C itself. This instruction set is not minimal, however, since the proper subset $\mathfrak{P}^{\rightarrow} \cup \mathfrak{J} \cup \{!\}$ suffices. This is demonstrated by the left behavior preserving endomorphism g, defined on individual C-instructions by

$$
\begin{aligned}
/a &\mapsto +/a; /\#2; /\#1, & \qquad \backslash a &\mapsto +/a; \backslash\#4; \backslash\#5,\\
+/a &\mapsto +/a; /\#2; /\#4, & \qquad +\backslash a &\mapsto +/a; \backslash\#4; \backslash\#8,\\
-/a &\mapsto +/a; /\#5; /\#1, & \qquad -\backslash a &\mapsto +/a; \backslash\#7; \backslash\#5,
\end{aligned}
$$

$$
\begin{aligned}
/\#k &\mapsto /\#3k; !; !, & \qquad \# &\mapsto /\#1; \backslash\#1; !,\\
\backslash\#k &\mapsto \backslash\#3k; !; !, & \qquad ! &\mapsto !; !; !.
\end{aligned}
$$

The next question which naturally arises is whether the instruction set $\mathfrak{P}^{\rightarrow} \cup \mathfrak{J} \cup \{!\}$ is minimal. For example, can we do with less than infinitely many jump instructions? And if not, will an infinite but otherwise arbitrary set of jump instructions suffice? We will now investigate those questions.

Recall the definition of the subsemigroup $C_{\leq k}$ in §5.3. As defined by (5.1), $C_{\leq k}$'s instruction set does not contain jump instructions with a jump counter greater than k.

Theorem 6.1 (Bergstra & Ponse)**.** *Let $|\mathcal{A}| \geq 2$. There does not exists a value $k \in \mathbb{N}^+$ such that $C_{\leq k}$ can express all finite threads.*

See the proof of Theorem 7 in [BP09a]; it has been replicated in Appendix B. See the proof of Theorem 6.2 below for a discussion.

Theorem 6.2. *Let $\mathcal{A}$ be non-empty. There does not exists a value $k \in \mathbb{N}^+$ such that $C_{\leq k}$ can express all finite threads.*

Proof. By Theorem 6.1 we conclude that if $|\mathcal{A}| \geq 2$, then $C_{\leq k}$ cannot express all finite threads. What remains is to be proved is that claim also holds if $|\mathcal{A}| = 1$. We do this by "patching" the proof by Bergstra & Ponse. As their proof is rather long we will not repeat it here—instead we summarize some key aspects of the proof, point out why it requires that $|\mathcal{A}| \geq 2$ and then proceed to show how this requirement can be eliminated. (Again, the proof is provided verbatim in Appendix B.)

The proof uses two key notions:

- Following the definition of residual threads by (2.1), the concept of *n-residual* threads is defined: Q is a 0-residual thread of P if $P = Q$. Q is an $(n{+}1)$-residual thread of P if $P = P_1 \trianglelefteq a \trianglerighteq P_2$ and Q is n-residual of either P_1 or P_2.
- Now a thread P has the *a-n-property* if $\pi_n(P) = a^n \circ \mathsf{D}$ and P has $2^n - 1$ distinct n-residuals with a first approximation not equal to $a \circ \mathsf{D}$.[1] An instruction sequence has the a-n-property if a thread with the a-n-property can be extracted from it.

[1] The sentences following this definition of the a-n-property in [BP09a] make it clear that P is meant to have 2^n instead of $2^n - 1$ distinct n-residuals with a first approximation not equal to $a \circ \mathsf{D}$. It turns out that this slightly weaker definition of the property does not affect the proof in any significant way.

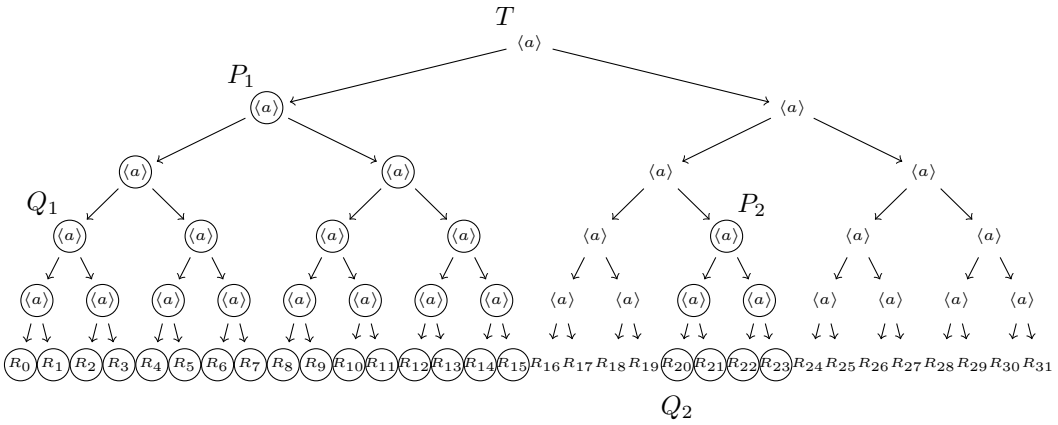

Figure 6.1: Graphical representation of a thread T with the a+5-property. The "leaves" R_n in this tree represent pairwise distinct 5-residuals of T which are each also distinct from any m-residual of T for $m < 5$. This in turns means that all m-residuals for $m \leq 5$ are pairwise distinct. For if e.g. P_1 and P_2 are not distinct, then Q_1 and Q_2 are identical as well, violating T's a+5-property. A similar argument holds for any pair of m-residuals with $m \leq 5$.

The proof by Bergstra and Ponse shows that for every $k \in \mathbb{N}$ there exists an $n \in \mathbb{N}^+$ such that no $C_{\leq k}$-expression X has the a-n-property. It does so by assuming the contrary and taking the minimal value for k in this respect. It is then shown that, given arbitrary $n \in \mathbb{N}^+$, one can find an $X \in \mathcal{I}^+_{C_{\leq k}}$ with the a-n-property for which it is also the case that $X \in \mathcal{I}^+_{C_{\leq k-1}}$. This contradicts the assumption that k was minimal.

Let P be a thread with the a-n-property. There are two observations to be made. First, if $n > 1$, then the set $\mathcal{A}$ of actions contains at least two elements, for otherwise the requirement that all first approximations of the distinct n-residuals of P must not equal $a \circ \mathsf{D}$ cannot be met.

Second, not only are all of P's n-residuals distinct, by extension the same holds of all m-residuals with $m < n$. Moreover, since all first approximations of n-residuals of P must not equal $a \circ \mathsf{D}$, it follows that for any m-residual Q and m'-residual R with $0 \leq m < m' \leq n$ it is necessarily so that $Q \neq R$.

Summarizing that second observation, we see that every m-residual ($m \leq n$) of a thread P with the a-n-property is unique. As a result any instruction sequence with the a-n-property has at least $2^n - 1$ distinct test instructions with action a.

Analyzing the proof, it turns out that it relies specifically on this second observation about threads with the a-n-property; requiring that threads with the a-n-property ($n > 1$) contain non-a actions is merely a means to that end. It turns out that we can define a slightly different class of threads with this second property without requiring that $|\mathcal{A}| \geq 2$: we say that a thread P has the a+n-property if $\pi_n(P) = a^n \circ \mathsf{D}$ and P has 2^n distinct n-residuals, none of which equals an $(n-m)$-residual of P (for $m > 0$).

To see why every m-residual ($m \leq n$) of a thread P with the a+n-property is unique, assume the contrary: then there are values m and m' with $m \leq m' \leq n$ such that some m-residual Q of P equals an m'-residual R of P. But then every $(n - m')$-residual of R equals some $(n - m')$-residual of Q. This yields a contradiction, because every $(n - m')$-residual of R is an n-residual of P, which is by definition distinct from any $(n - m')$-residual of Q, because $m + (n - m') \leq n$. Figure 6.1 attemps to visualize this argument using a thread T with the a+5-property.

For every $n \in \mathbb{N}^+$ a thread P with the a+n-property can be created, such that P performs only a actions. Fix some n and let $g\colon [0, 2^n - 1] \to \{\mathtt{true}, \mathtt{false}\}^n$ be a bijection, where

$\{\texttt{true}, \texttt{false}\}^n$ is the set of all boolean sequences of length n. We write $(g(m))_{d+1}$ for the $(d{+}1)$th element of $g(m)$. Now we define the family of threads P^l for all $1 \leq l < 2^n$ such that:[2]

$$P^l = \begin{cases} P^{2l} \trianglelefteq a \trianglerighteq P^{2l+1} & \text{if } l < 2^{n-1}, \\ Q^n_{2l-2^n} \trianglelefteq a \trianglerighteq Q^n_{2l-2^n+1} & \text{otherwise,} \end{cases} \tag{6.1a}$$

$$Q^0_m = a \circ \mathsf{D}, \tag{6.1b}$$

$$Q^{d+1}_m = \begin{cases} Q^d_m \trianglelefteq a \trianglerighteq \mathsf{D} & \text{if } (g(m))_{d+1} = \texttt{false}, \\ \mathsf{D} \trianglelefteq a \trianglerighteq Q^d_m & \text{otherwise.} \end{cases} \tag{6.1c}$$

Informally, the thread P^1 performs n a-actions after which some state Q^n_m is reached. Due to the nature of g, $Q^n_m \neq Q^n_{m'}$ for distinct m and m'. For example, for $n = 2$ and taking g such that

$$0 \mapsto \{\texttt{false}, \texttt{false}\}, \quad 1 \mapsto \{\texttt{false}, \texttt{true}\}, \quad 2 \mapsto \{\texttt{true}, \texttt{false}\}, \quad 3 \mapsto \{\texttt{true}, \texttt{true}\},$$

the following equations are defined:

$$P^1 = P^2 \trianglelefteq a \trianglerighteq P^3, \qquad P^2 = Q^2_0 \trianglelefteq a \trianglerighteq Q^2_1, \qquad P^3 = Q^2_2 \trianglelefteq a \trianglerighteq Q^2_3,$$

and,

$$\begin{array}{llll} Q^2_0 = Q^1_0 \trianglelefteq a \trianglerighteq \mathsf{D}, & Q^2_1 = Q^1_1 \trianglelefteq a \trianglerighteq \mathsf{D}, & Q^2_2 = \mathsf{D} \trianglelefteq a \trianglerighteq Q^1_2, & Q^2_3 = \mathsf{D} \trianglelefteq a \trianglerighteq Q^1_3, \\ Q^1_0 = Q^0_0 \trianglelefteq a \trianglerighteq \mathsf{D}, & Q^1_1 = \mathsf{D} \trianglelefteq a \trianglerighteq Q^0_1, & Q^1_2 = Q^0_2 \trianglelefteq a \trianglerighteq \mathsf{D}, & Q^1_3 = \mathsf{D} \trianglelefteq a \trianglerighteq Q^0_3, \\ Q^0_0 = a \circ \mathsf{D}, & Q^0_1 = a \circ \mathsf{D}, & Q^0_2 = a \circ \mathsf{D}, & Q^0_3 = a \circ \mathsf{D}. \end{array}$$

Observe that any thread Q^n_m performs $n+1$ a-actions only if the sequence of boolean replies yielded by the first n actions is exactly according to $g(m)$. Thus each thread Q^n_m is a unique n-residual of P^1 (recall that g is bijective). Since D is a 1-residual of every thread Q^n_m, but not of any thread P^l we conclude that P^1 meets the necessary criteria to have the $a{+}n$-property.

Replacing any thread with the a-n-property in the proof of Bergstra & Ponse with a thread with the $a{+}n$-property results in a valid proof which requires only that $|\mathcal{A}| \neq \emptyset$, as opposed to $|\mathcal{A}| > 1$. This proves our claim. □

We have now established that arbitrarily many distinct jump instructions are required to let C express all finite threads. It turns out that jump instructions in a single direction suffice.

Proposition 6.3. *Let $J^{\rightarrow} \subseteq \mathfrak{J}^{\rightarrow}$ be an infinite but otherwise arbitrary set of forward jump instructions and let the code semigroup C' be generated by the instruction set $\mathfrak{P}^{\rightarrow} \cup J^{\rightarrow} \cup \{!\}$. Then C' can express all finite thread but no infinite threads. This also holds if $\mathfrak{P}^{\rightarrow}$ is replaced by $\mathfrak{N}^{\rightarrow}$. If $J^{\leftarrow} \subseteq \mathfrak{J}^{\leftarrow}$ is an infinite but otherwise arbitrary set of backward jump instructions, then the instruction sets $\mathfrak{P}^{\leftarrow} \cup J^{\leftarrow} \cup \{!\}$ and $\mathfrak{N}^{\leftarrow} \cup J^{\leftarrow} \cup \{!\}$ also generate a code semigroup which characterizes BTA.*

Proof. As C' does not contain backward instructions, it cannot create any kind of loop (for all $i, j \in \mathbb{Z}$, if $i \rightarrow_X j$ according to some $X \in \mathcal{I}^+_{C'}$, then necessarily $i < j$). Every instruction sequence is finite, thus so is any thread extracted from a C'-inseq X. What remains to be shown is that all BTA threads can be described by C'.

Let $P \in \mathrm{BTA}$ be a finite thread. We will inductively construct a C' instruction sequence X_P such that $|X_P|^{\rightarrow}_C = P$. For convenience we will define $F = \{\delta(u) \mid u \in J^{\rightarrow}\}$ to be the set of jump counters of admitted jump instructions.

[2] In this definition relevant values for d and m are in the ranges $[0, n-1]$ and $[0, 2^n - 1]$, respectively.

If $P = \mathsf{S}$ then define $X_P = !$. If $P = \mathsf{D}$ then define $X_P = /\#k$, for some $k \in F$. Otherwise $P = Q \trianglelefteq a \trianglerighteq R$ for some $a \in \mathcal{A}$ and $Q, R \in \mathrm{BTA}$. By induction there are $X_Q, X_R \in \mathcal{I}^+_{C'}$ such that $|X_Q|^{\rightarrow}_C = Q$ and $|X_R|^{\rightarrow}_C = R$.

Create an inseq X'_R from X_R by changing the jump counter k of any jump instruction at an exit position in X_R to some value $k' \in \{j \in F \mid j \geq k + \ell(X_Q)\}$. (These are the instructions which upon execution cause deadlock).

Now we define $X_P = +/a; /\#k; X'_R; (!)^p; X_Q$, where $k \in \{j \in F \mid j > \ell(X'_R)\}$ and $p = k - \ell(X'_R) - 1$. It is not hard to see that indeed $|X_P|^{\rightarrow}_C = P$. Note that the termination instructions introduced here are solely for the purpose of *padding*. They are not reachable from the leftmost instruction.

A similar construction can be made using negative tests. When using backward jump instructions create an inseq X_P such that $|X_P|^{\leftarrow}_C = P$. □

Although all finite threads can be expressed using jump instructions in only one direction, this is not the case for all regular threads. In fact, infinitely many distinct jump instructions in both directions are necessary.

Definition 6.4. In an instruction sequence $X = u_1; u_2; \ldots; u_k \in \mathcal{I}^+_A$ an instruction u_j is *i-n-relevant* if there exists an instruction sequence X', created from X by changing u_j to some other instruction $u \in \mathcal{I}_A$, such that $\pi_n(|X|^i_A) \neq \pi_n(|X'|^i_A)$. In other words: the nth projection of the execution of inseq X starting at position i depends on u_j. Observe that any instruction which is i-n-relevant is also i-$(n{+}1)$-relevant.

Theorem 6.5. *Let $\mathcal{A}$ be non-empty and fix some $k \in \mathbb{N}^+$. Let $\mathcal{I}_{C'}$ be the largest subset of $\mathcal{I}_C$ which does not contain forward (backward) jump instructions with a jump counter greater than k (i.e., $\mathcal{I}_{C'}$ contains a finite number of forward or backward jump instructions). Then the semigroup C' generated by $\mathcal{I}_{C'}$ cannot express all regular threads.*

Proof. Let k be fixed and select n such that $2^n \geq 2k + 3$. We will assume that C' restricts forward jump instructions (a similar argument holds if backward jump instructions are restricted). Let $g\colon [0, 2^{2n} - 1] \to \{\mathtt{true}, \mathtt{false}\}^{2n}$ be a bijection, where $\{\mathtt{true}, \mathtt{false}\}^{2n}$ is the set of all boolean sequences of length $2n$. We write $(g(m))_{d+1}$ for the $(d{+}1)$th element of $g(m)$. Now we define the family of threads P^l for all $1 \leq l < 2^{2n}$ such that:[3]

$$P^l = \begin{cases} P^{2l} \trianglelefteq a \trianglerighteq P^{2l+1} & \text{if } l < 2^{2n-1}, \\ Q^{2n}_{2l-2^{2n}} \trianglelefteq a \trianglerighteq Q^{2n}_{2l-2^{2n}+1} & \text{otherwise,} \end{cases} \tag{6.2a}$$

$$Q^0_m = \mathsf{D}, \tag{6.2b}$$

$$Q^{d+1}_m = \begin{cases} Q^d_m \trianglelefteq a \trianglerighteq P^{2^n + [m]_{2^n}} & \text{if } (g(m))_{d+1} = \mathtt{false}, \\ P^{2^n + [m]_{2^n}} \trianglelefteq a \trianglerighteq Q^d_m & \text{otherwise.} \end{cases} \tag{6.2c}$$

Figure 6.2 presents a graphical representation of thread P^1 for $n = 2$. Observe the similarities of this set of equations to those presented in (6.1). Recall from §5.3.1 that $[m]_{2^n}$ is the remainder of m after division by 2^n. Informally, the thread P^1 performs $2n$ a-actions after which some state Q^{2n}_m is reached. Distinct sequences of boolean replies to these actions result in distinct values for m ($0 \leq m < 2^{2n}$). Due to the nature of g, $Q^{2n}_m \neq Q^{2n}_{m'}$ for distinct m and m'. (To see why, observe that the $2n$-residual D of Q^{2n}_m can be reached starting in state Q^{2n}_m only if the replies to the first $2n$ a-actions are precisely according to $g(m)$—and g is a bijection). Thus each thread Q^{2n}_m is a unique $2n$-residual of P^1. Since D is a $2n$-residual of every thread Q^{2n}_m, but not of any thread P^l we conclude that P^1 meets the necessary criteria to have the $a{+}2n$-property.

Towards a contradiction assume that there exists a C'-expression X such that $|X|^i_C = P^1$ for some $i \in [1, \ell(X)]$. We define $f(l) = \min\{i \mid |X|^i_C = P^l\}$ to be the function which

[3] In this definition relevant values for d and m are in the ranges $[0, 2n-1]$ and $[0, 2^{2n}-1]$, respectively.

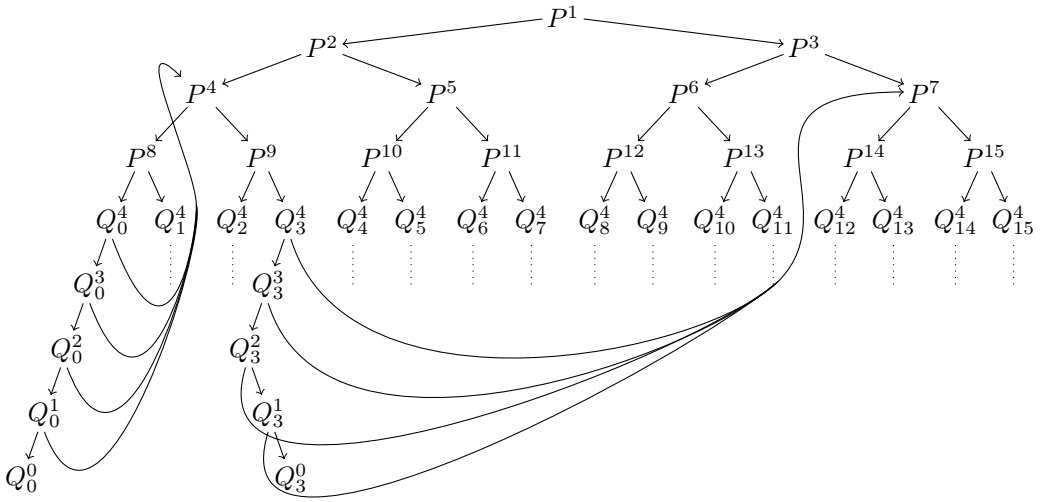

Figure 6.2: Graphical representation of the thread described by P^1 as defined by (6.2), for $n = 2$. Observe that the threads P^4, P^5, P^6 and P^7 (i.e. the threads P^{2^n} through $P^{2^{n+1}-1}$) are n-residuals of of P^1. Likewise each thread thread $Q_m^4 = Q_m^{2n}$ is a $2n$-residual of P^1. Each thread Q_m^4 is distinct, and each of P^1's n-residuals is a residual thread of each thread Q_m^4. Expanded are threads Q_0^4 and Q_3^4 which are defined according to $g(0) = \{\texttt{false}, \texttt{false}, \texttt{false}, \texttt{false}\}$ and $g(3) = \{\texttt{false}, \texttt{false}, \texttt{true}, \texttt{true}\}$, respectively. Note that $Q_m^0 = \mathsf{D}$ for all $m \in [0, 15]$, thus in particular $Q_0^0 = Q_3^0 = \mathsf{D}$.

returns the leftmost position in X from which the thread P^l can be extracted. Without loss of generality we will assume that all instructions in X are reachable from position i, for if not, then by Proposition 3.5 we can create an instruction sequence X' for which this *does* hold. The largest jump counter of any forward jump instruction in X' would be less than or equal to the largest forward jump distance in X.

For distinct $l, l' < 2^{2n}$ it is the case that $P^l \neq P^{l'}$ (because P^1 has the $a{+}2n$-property) and thus necessarily $f(l) \neq f(l')$. The n-residuals of P^1 are the threads P^l for $l \in [2^n, 2^{n+1} - 1]$. The integers in this range are totally ordered by the function f:[4]

$$l_0, l_1, \ldots, l_{2^n-1}.$$

No instruction in X is both $f(l_i)$-n-relevant and $f(l_j)$-n-relevant for distinct i and j, because every thread P^{l_i} is an n-residual of P^1, and P^1 has the $a{+}2n$-property. Moreover, the n-residuals of any thread P^{l_i} are the threads $Q^{2n}_{i2^n+m}$, for $0 \leq m < 2^n$. The thread P^{l_m} in turn is an 1-residual (and a $2, 3, \ldots, 2n$-residual) of the thread $Q^{2n}_{i2^n+m}$. Thus every thread P^{l_j} is a residual thread of every thread P^{l_i}.

Recall that $2^n \geq 2k + 3$ and that C' does not contain forward jump instructions over a distance greater than k. Thus for some $i < k + 1$ all $f(l_i)$-n-relevant instructions are left of position $f(l_{k+1})$. For if not, then there are $k{+}1$ distinct positions $< f(l_{k+1})$ containing jump instructions which target $k + 1$ distinct positions $> f(l_{k+1})$. This is not possible because of the restriction on forward jump counters.

Fix said i, and note that there are at least $k + 1$ instructions which are $f(l_i)$-$(n{+}1)$-relevant to the right of $f(l_{k+1})$: namely $f(l_{k+2}), f(l_{k+3}), \ldots, f(l_{2k+2})$. This leads to a contradiction, since this, too, is not possible because of the restriction on jump counters. □

[4]The ordering on $[2^n, 2^{n+1} - 1]$ imposed by f does not need to be the natural ordering of these integers!

Now that it has been established that an upper bound on the value of jump counters limits expressiveness, even if only in a single direction, the question naturally arises whether *any* two infinite collections of forward and backward jump instructions suffice to express all regular threads. We prove that this is indeed the case.

Theorem 6.6. *Let $J^{\rightarrow} \subseteq \mathfrak{J}^{\rightarrow}$ and $J^{\leftarrow} \subseteq \mathfrak{J}^{\leftarrow}$ be two infinite but otherwise arbitrary sets of jump instructions and let the code semigroup C' be generated by the set $\mathfrak{P}^{\rightarrow} \cup J^{\rightarrow} \cup J^{\leftarrow} \cup \{!\}$. Then all regular threads can be expressed by C'. This also holds if $\mathfrak{P}^{\rightarrow}$ is replaced by $\mathfrak{N}^{\rightarrow}$, $\mathfrak{P}^{\leftarrow}$ or $\mathfrak{N}^{\leftarrow}$.*

Proof. Fix some infinite $J^{\rightarrow} \subseteq \mathfrak{J}^{\rightarrow}$ and $J^{\leftarrow} \subseteq \mathfrak{J}^{\leftarrow}$ and select arbitrary $T \in \mathrm{BTA}^{\mathrm{reg}}$ with states $P_0, P_1, \ldots, P_{n-1}$. Then the result of the procedure ConstructInseq(T, $\{\delta(u) \mid u \in J^{\rightarrow}\}$, $\{\delta(u) \mid u \in J^{\leftarrow}\}$) as outlined in Algorithm 6.1 is a C'-inseq X such that $|X|_C^{\rightarrow} = T$.

Algorithm 6.1 C-expression construction using a restricted set of jump counters

Require: A regular thread T with states $P_0, P_1, \ldots, P_{n-1}$ and infinite sets $F, B \subseteq \mathbb{N}$.
Ensure: A C-inseq X with $|X|_C^{\rightarrow} = P_0$, $\{\delta(u) \mid u \in \mathfrak{J}^{\rightarrow}(X)\} \subset F$, $\{\delta(u) \mid u \in \mathfrak{J}^{\leftarrow}(X)\} \subset B$.

```
 1: procedure ConstructInseq(T, F, B)
 2:     s ← RandomSelect({j ∈ F | j ≥ 4})
 3:     z ← n · s · (s − 1)                          ▷ Largest (rightmost) instruction position
 4:     I ← ∅                                        ▷ Set of (position, instruction) tuples
 5:     for i ← 0 to n − 1 do
 6:         for r ← 0 to s − 1 do
 7:             c ← (i · s + r) · (s − 1) + 1
 8:             if P_i = S then
 9:                 I ← I ∪ {(c, !)}
10:             else if P_i = D then
11:                 d ← RandomSelect({j ∈ B | j ≥ c})
12:                 I ← I ∪ {(c, \#d)}               ▷ Jump outside program: deadlock
13:             else if P_i = P_j ⊴ a ⊵ P_k then
14:                 I ← I ∪ {(c, +/a)}
15:                 I ← I ∪ Connect(c + 1, j · s · (s − 1) + 1, z, s, F, B)
16:                 z ← max{p | ∃u[(p, u) ∈ I]}
17:                 I ← I ∪ Connect(c + 2, k · s · (s − 1) + 1, z, s, F, B)
18:                 z ← max{p | ∃u[(p, u) ∈ I]}
19:             end if
20:         end for
21:     end for
22:     return ConcatInstructions(I ∪ {(p, !) | 0 < p < z, ¬∃u[(p, u) ∈ I]})
23: end procedure

24: procedure Connect(i, j, z, s, F, B)
25:     r ← i + RandomSelect({k ∈ F | i + k > z})
26:     l ← r − RandomSelect({k ∈ B | r − k ≤ j})
27:     p ← ⌊(j − l)/s⌋
28:     p ← p + j − (l + p · s)
29:     return {(i, /\#r−i)} ∪ {(r + k · s, /\#s) | 0 ≤ k < p} ∪ {(r + p · s, \#r−l)}
30: end procedure
```

Suppose we want to transfer control of execution in an inseq X from position i to position j. Obviously, $J^{\rightarrow} \cup J^{\leftarrow}$ may not contain the jump instruction required to jump immediately from i to j. In fact, it may be so that no sequence of jump instructions permitted by $J^{\rightarrow} \cup J^{\leftarrow}$ can transfer control of execution from position i to j. For example, if only even

jump counters are available, then control of execution cannot be transferred from i to j if $i - j$ is odd.

Algorithm 6.1 solves this issue by producing an instruction sequence X in which functionally equivalent subsequences of instructions are repeated s times at evenly spaced intervals of length $s-1$. The value of s is selected from the set of permissible forward jump counters $J^{\rightarrow}$, with the sole restriction that $s \geq 4$. Thus, for any P_k there are at least s positions $j_0, j_1, \ldots, j_{s-1}$ (with $j_{m+1} = j_m + s - 1$) in X from which P_k can be extracted and for any position i in X there is at least one such position j_m such that $i = j_m \pmod{s}$.

Now the general procedure to "connect" a position i to one such j_m in X using a sequence of permissible jump instructions is to extend X with a sequence of jump instructions to the right of X, as follows. First, select a sufficiently large forward jump instruction f which, if placed at position i, jumps outside of X to some position r. Second, select a sufficiently large backward jump instruction b which, if placed at position r, jumps to a position $l \leq j_0$. Now observe that, instead of placing b at position r, we can add a sequence of chained $/\#s$ instructions, starting at position r and extending to the right, such that they transfer control of execution to some position $r' > r$. r' can be selected such that if the backward instruction b were placed there, it would jump to a position l' between $j_0 - (s-1)$ and j_0. By adding another $j_0 - l'$ chained $/\#s$ instructions starting at position r', control of execution will be transferred to a position $r'' > r'$ from which the instruction b will target exactly one of the positions j_m. Specifically, $m = j_0 - l'$. The procedure described here is performed by $\textsc{Connect}(i,\, j_1,\, \ell(X),\, s,\, \{\delta(u) \mid u \in J^{\rightarrow}\},\, \{\delta(u) \mid u \in J^{\leftarrow}\})$, which returns the required jump instructions and the positions where they should be placed.

The procedure $\textsc{ConstructInseq}(T,\, \{\delta(u) \mid u \in J^{\rightarrow}\},\, \{\delta(u) \mid u \in J^{\leftarrow}\})$ selects a suitable value s and ensures that for every thread P_i there are s positions $j_0, j_1, \ldots, j_{s-1}$ from which P_i can be extracted. At each of these positions it places a suitable instruction: ! if $P_i = \mathsf{S}$, # if $P_i = \mathsf{D}$ and $+/a$ if $P_i = P_{i'} \trianglelefteq a \trianglerighteq P_{i''}$. In the latter case $\textsc{Connect}(\ldots)$ is used to ensure that indeed either of $P_{i'}$ and $P_{i''}$ will be reached after exectution of action a. □

6.2 The Expressiveness of Subsemigroups of Cg

Equipped with the translations of Chapter 5 and the theorems of §6.2, we are now ready to make statements about the expressiveness of Cg and some of its subsemigroups.

Proposition 6.7. *Each thread definable in Cg is regular, and each regular thread can be expressed in Cg.*

Proof. This follows immediately from the fact that C2CG and CG2C are behavior preserving and total. Since C characterizes the regular threads (see Proposition 3.1), so does Cg. □

Theorem 6.8. *Let $\mathcal{A}$ be non-empty. There does not exists a value $k \in \mathbb{N}^+$ such that $Cg_{\leq k}$ can express all finite threads.*

Proof. Upon analyzing the family of translations CG2C_k as defined in §5.4.1, we see that they map $Cg_{\leq k}$-expressions to behaviorally equivalent $C_{\leq 4k+12}$-expressions.

Thus if $Cg_{\leq k}$ can express all finite threads, then so can $C_{\leq 4k+12}$. But by Theorem 6.2 this is impossible. □

Proposition 6.9. *Let $G^{\rightarrow} \subseteq \mathfrak{G}^{\rightarrow}$ be an infinite but otherwise arbitrary set of forward goto instructions and let $L^{\rightarrow} \subseteq \mathfrak{L}^{\rightarrow}$ constitute the set of label instructions which match the goto instructions in $G^{\rightarrow}$. Then the code semigroup Cg' generated by the instruction set $\mathfrak{P}^{\rightarrow} \cup G^{\rightarrow} \cup L^{\rightarrow} \cup \{!\}$ can express all finite threads but no infinite threads. This also holds if $\mathfrak{P}^{\rightarrow}$ is replaced by $\mathfrak{N}^{\rightarrow}$. If the infinite sets $G^{\leftarrow} \subseteq \mathfrak{G}^{\leftarrow}$ and $L^{\leftarrow} \subseteq \mathfrak{L}^{\leftarrow}$ are defined analogously, then the instruction sets $\mathfrak{P}^{\leftarrow} \cup G^{\leftarrow} \cup L^{\leftarrow} \cup \{!\}$ and $\mathfrak{N}^{\leftarrow} \cup G^{\leftarrow} \cup L^{\leftarrow} \cup \{!\}$ also generate a code semigroup capable of expressing all finite threads.*

Proof. As in the proof of Proposition 6.3 we observe that Cg' does not contain backward instructions. Thus it can only express finite threads, as loops (a requirement for infinite behavior) cannot be constructed in Cg'. Now we need to show all BTA threads can be expressed by Cg'.

We will inductively define a Cg' instruction sequence X_P for every $P \in \mathrm{BTA}$ such that $|X_P|^{\rightarrow}_{Cg} = P$. Let $F = \{\lambda(u) \mid u \in G^{\rightarrow}\}$ be the set of label numbers of available goto instructions.

If $P = \mathsf{S}$ then $X_P = !$. If $P = \mathsf{D}$ then set $X_P = /\#\#£l$, where l is an arbitrary element of F. Otherwise $P = Q \trianglelefteq a \trianglerighteq R$ and there are $X_Q, X_R \in \mathcal{I}^{+}_{Cg'}$ such that $|X_Q|^{\rightarrow}_{Cg} = Q$ and $|X_R|^{\rightarrow}_{Cg} = R$. Select some label number $l \in F$ such that it is not present in X_Q or X_R. Then $X_P = +/a; /\#\#£l; X_R; /£l; X_Q$.

A similar construction can be made using negative tests. When using backward goto instructions create an inseq X_P such that $|X_P|^{\leftarrow}_{Cg} = P$. □

Theorem 6.10. *Let $\mathcal{A}$ be non-empty and fix some value $k \in \mathbb{N}^{+}$. Let $\mathcal{I}_{Cg'}$ be the largest subset of $\mathcal{I}_{Cg}$ which does not contain forward (backward) goto instructions with a label number k or greater (i.e., $\mathcal{I}_{Cg'}$ contains a finite number of forward or backward goto instructions). Then the semigroup Cg' generated by $\mathcal{I}_{Cg'}$ cannot express all regular threads.*

Proof. The proof is analogous to that of Theorem 6.5. Again select n such that $2^n \geq 2k+3$ and consider the thread P_1 as defined by (6.2). As before the function $f(l) = \min\{i \mid |X|^{i}_{C} = P^{l}\}$ induces a total ordering on the range $[2^n, 2^{n+1}-1]$, say $l_0, l_1, \ldots l_{2^n-1}$. Observe that for some $i < k+1$ all $f(l_i)$-n-relevant instructions are left of position $f(l_{k+1})$, for otherwise there must be $k+1$ distinct goto instructions on positions $< f(l_{k+1})$ which target $k+1$ distinct label instructions on positions $> f(l_{k+1})$; impossible, as $\mathcal{I}_{Cg'}$ contains only k distinct forward goto instructions.

Fixing said i we note that there are at least $k+1$ positions which are $f(l_i)$-$(n+1)$-relevant to the right of $f(l_{k+1})$: this too is impossible, for the same reason. Contradiction. □

Chapter 7

Discussion

This thesis can be divided into four parts: the introduction of C and the theory behind it, the introduction of Cg as an alternative to C, the definition of translations between these, and several results about the expressiveness of C and Cg.

We have proved that C and Cg are equally expressive by means of the total mappings c2cg and cg2c. We have also proved that such translations are only possible if the maximum jump counter (or label number) in the input inseq is known. As a result c2cg and cg2c cannot be homomorphic.

We then went on to prove that any subsemigroup of C (Cg) needs to contain infinitely many jump instructions (matching label and goto instructions) in order to express all finite threads (Theorem 6.2, Theorem 6.8). In order to express all regular threads it is even necessary that such a semigroup contains infinitely many jump instructions (label/goto instructions) *in both directions* (Theorem 6.5, Theorem 6.10). The upshot is that any such infinite collection of jump instructions (label/goto instructions) suffices (Theorem 6.6, the corresponding result for Cg is trivial).

7.1 Further Work

The translations between C and Cg in Chapter 5 use label and goto instructions to mimic the behavior of jump instructions and vice versa. There are some open questions about the nature of these translations: it is not known whether alternative behavior preserving mappings can be defined which employ less jump instructions or label/goto instructions. More precisely,

- Given an arbitrary value $k \in \mathbb{N}$, what is the smallest value $k' \in \mathbb{N}$ for which there exists a behavior preserving mapping $f\colon \mathcal{I}^+_{Cg_{\leq k}} \to \mathcal{I}^+_{C_{\leq k'}}$? (By definition of equation (5.8) in §5.4.1 we already know that $k' \leq 4k + 12$.)
- As demonstrated by the translations defined in §5.3, there exist behavior preserving mappings $f\colon \mathcal{I}^+_{C_{\leq k}} \to \mathcal{I}^+_{Cg_{\leq k}}$ for all $k > 2$. Is there any value $k \in \mathbb{N}$ such that for some $k' < k$ the mapping $f\colon \mathcal{I}^+_{C_{\leq k}} \to \mathcal{I}^+_{Cg_{\leq k'}}$ is behavior preserving?

7.2 Acknowledgements

First and foremost I want to thank my supervisor, Alban Ponse, for his guidance and most of all patience; the writing of this thesis took much longer than it should have. I thank Kyndylan Nienhuis for asking some smart questions about the semantics of Cg, which led to the inclusion of §4.6.

I thank my family and especially Vera Matei for their support during the writing of this thesis.

Appendix **A**

Overview of Defined Translations

Figure A.1 provides a graphical representation of the most important sets of (single pass) instruction sequences introduced in this thesis. Recall that the set $\mathcal{I}^+_C$ contains all C-expressions. For arbitrary $k \in \mathbb{N}$, $\mathcal{I}^+_{C_{\leq k}}$ is the largest subset of $\mathcal{I}^+_C$ which does not contain C-inseqs with relative jumps over a distance greater than k. Similarly, $\mathcal{I}^+_{Cg}$ contains all Cg-expressions, and $\mathcal{I}^+_{Cg_{\leq k}}$ contains those inseqs without goto instructions with a label number greater than k. All PGA terms are contained in $\boldsymbol{P}$; the set $\boldsymbol{P_1}$ is the largest set which is restricted to single pass instruction sequences in first canonical form. $\boldsymbol{P_2}$ contains PGA's second canonical forms.

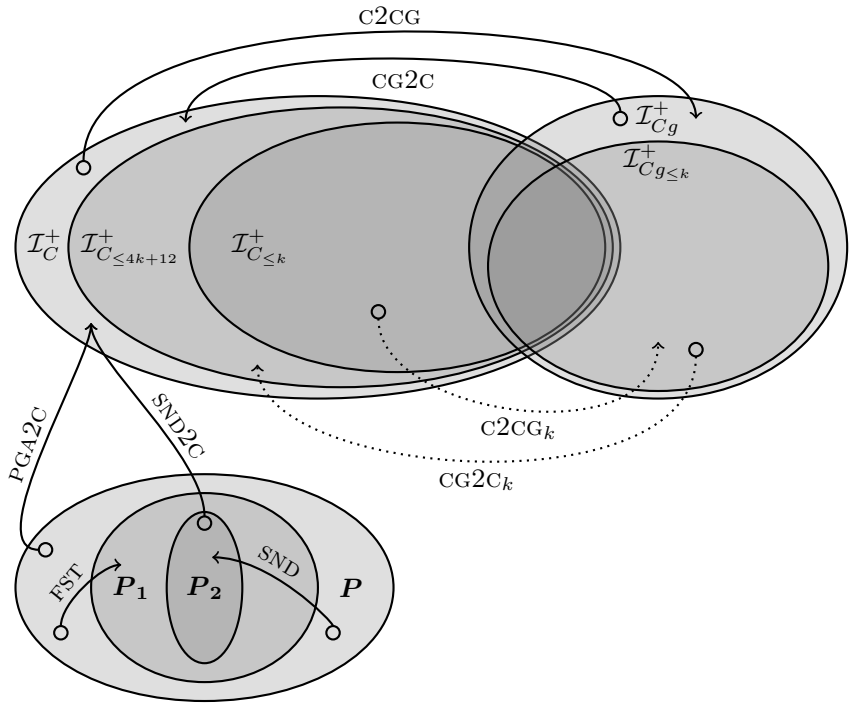

Figure A.1: Overview of semigroups and single-pass instruction sequences and certain behavior preserving mappings defined between then, as introduced in this thesis. Dotted arrows represent homomorphisms. There is also a non-homomorphic version of C2CG$_k$ (§5.3.1).

Appendix **B**

Proof by Bergstra & Ponse

The proof of Theorem 6.1 is presented in Section 9 of [BP09a]. As the proof of Theorem 6.2 builds upon this result, Section 9 of [BP09a] is reproduced here verbatim, with kind permission of the authors. Three minor changes have been applied: a section reference has been updated to point to an equivalent section in this thesis, a footnote has been added and the last paragraph has been left out, as it is merely an introduction to Section 10 of that publication.

Observe that [BP09a] uses notation which in some places differs slightly from notation introduced in this thesis.

B.1 Expressiveness and reduced instruction sets

In this section we further consider C's instructions in the perspective of expressiveness. We show that setting a bound on the size of jump counters in C does have consequences with respect to expressiveness: let

$$C_k$$

be defined by allowing only jump instructions with counter value k or less.

We first introduce some auxiliary notions: following the definition of residual threads in Section 2.1, we say that thread Q is a 0-*residual* of thread P if $P = Q$, and an $n+1$-*residual* of P if for some $a \in A$, $P = P_1 \trianglelefteq a \trianglerighteq P_2$ and Q is an n-residual of P_1 or of P_2. Note that a finite thread (in BTA) only has n-residuals for finitely many n, while for the thread P defined by $P = a \circ P$ it holds that P is an n-residual of itself for each $n \in \mathbb{N}$.

Let $a \in A$ be fixed and $n \in \mathbb{N}^+$. Thread P has the *a-n-property* if $\pi_n(P) = a^n \circ \mathsf{D}$ and P has $2^n - 1$ (different) n-residuals which all have a first approximation not equal to $a \circ \mathsf{D}$.[1] So, if a thread P has the a-n-property, then n consecutive a-actions can be executed and each sequence of n replies leads to a unique n-residual. Moreover, none of these residual threads starts with an a-action (by the requirement on their first approximation). We note that for each $n \in \mathbb{N}^+$ we can find a finite thread with the a-n-property. In the next section we return to this point.

A piece of code X has the a-n-property if for some i, $|X|_i$ has this property. It is not hard to see that in this case X contains at least $2^n - 1$ different a-tests.

[1] It appears that the authors meant to use 2^n instead of $2^n - 1$ in this sentence, though this does not affect the proof in any serious way. —Stephan

As an example, consider

$$X = !\,;\backslash b; +\backslash a; +/a; \backslash\#2; +/a; /\#2; /c; \#$$

Clearly, X has the a-2-property because $|X|_4$ has this property: its 2-residuals are $b \circ \mathsf{S}$, S, D and $c \circ \mathsf{D}$, so each thread is not equal to one of the others and does not start with an a-action.

Note that if a piece of code X has the a-$(n+k)$-property, then it also has the a-n-property. In the example above, X has the a-1-property because $|X|_3$ has this property (and $|X|_6$ too).

Lemma 1. For each $k \in \mathbb{N}$ there exists $n \in \mathbb{N}^+$ such that no $X \in C_k$ has the a-n-property.

Proof. Suppose the contrary and let k be minimal in this respect. Assume for each $n \in \mathbb{N}^+$, $Y_n \in C_k$ has the a-n-property.

Let $B = \{\mathtt{true}, \mathtt{false}\}$. For $\alpha, \beta \in B^*$ we write

$$\alpha \preceq \beta$$

if α is a prefix of β, and we write $\alpha \prec \beta$ or $\beta \succ \alpha$ if $\alpha \preceq \beta$ and $\alpha \neq \beta$. Furthermore, let

$$B^{\leq n} = \bigcup_{i=0}^{n} B^i,$$

thus $B^{\leq n}$ contains all B^*-sequences α with $\ell(\alpha) \leq n$ (there are $2^{n+1} - 1$ such sequences).

Let $g : \mathbb{N} \to \mathbb{N}$ be such that $|Y_n|_{g(n)}$ has the a-n-property. Define

$$f_n : B^{\leq n} \to \mathbb{N}^+$$

by $f_n(\alpha) = m$ if the instruction reached in Y_n when execution started at position $g(n)$ after the replies to a according to α has position m. Clearly, f_n is an injective function.

In the following claim we show that under the supposition made in this proof a certain form of squeezing holds: if k' is sufficiently large, then for all $n > 0$ there exist $\alpha, \beta, \gamma \in B^{k'}$ with $f_{k'+n}(\alpha) < f_{k'+n}(\beta) < f_{k'+n}(\gamma)$ with the property that $f_{k'+n}(\alpha) < f_{k'+n}(\beta') < f_{k'+n}(\gamma)$ for each extension β' of β within $B^{\leq k'+n}$. This claim is proved by showing that not having this property implies that "too many" such extensions β' exist. Using this claim it is not hard to contradict the minimality of k.

Claim 1. Let k' satisfy $2^{k'} \geq 2k+3$. Then for all $n > 0$ there exist $\alpha, \beta, \gamma \in B^{k'}$ with

$$f_{k'+n}(\alpha) < f_{k'+n}(\beta) < f_{k'+n}(\gamma)$$

such that for each extension $\beta' \succeq \beta$ in $B^{\leq k'+n}$,

$$f_{k'+n}(\alpha) < f_{k'+n}(\beta') < f_{k'+n}(\gamma).$$

Proof of Claim 1. Let k' satisfy $2^{k'} \geq 2k+3$. Towards a contradiction, suppose the stated claim is not true for some $n > 0$. The sequences in $B^{k'}$ are totally ordered by $f_{k'+n}$, say

$$f_{k'+n}(\alpha_1) < f_{k'+n}(\alpha_2) < \ldots < f_{k'+n}(\alpha_{2^{k'}}).$$

Consider the following list of sequences:

$$\alpha_1, \underbrace{\alpha_2, \ldots, \alpha_{2k+2}}_{\text{choices for } \beta}, \alpha_{2k+3}$$

By supposition there is for each choice $\beta \in \{\alpha_2, \ldots, \alpha_{2k+2}\}$ an extension $\beta' \succ \beta$ in $B^{\leq k'+n}$ with

$$\text{either} \quad f_{k'+n}(\beta') < f_{k'+n}(\alpha_1), \quad \text{or} \quad f_{k'+n}(\beta') > f_{k'+n}(\alpha_{2k+3}).$$

Because there are $2k+1$ choices for β, assume that at least $k+1$ elements $\beta \in \{\alpha_2, \ldots, \alpha_{2k+2}\}$ have an extension β' with

$$f_{k'+n}(\beta') < f_{k'+n}(\alpha_1)$$

(the assumption $f_{k'+n}(\beta') > f_{k'+n}(\alpha_{2k+3})$ for at least $k+1$ elements β with extension β' leads to a similar argument). Then we obtain a contradiction with respect to $f_{k'+n}$: for each of the sequences β in the subset just selected and its extension β',

$$f_{k'+n}(\beta') < f_{k'+n}(\alpha_1) < f_{k'+n}(\beta),$$

and there are at least $k+1$ different such pairs β, β' (recall $f_{k'+n}$ is injective). But this is not possible with jumps of at most k because the $f_{k'+n}$ values of each of these pairs define a path in $Y_{k'+n}$ that never has a gap that exceeds k and that passes position $f_{k'+n}(\alpha_1)$, while different paths never share a position. This finishes the proof of Claim 1. □

Take according to Claim 1 an appropriate value k', some value $n > 0$ and $\alpha, \beta, \gamma \in B^{k'}$. Consider $Y_{k'+n}$ and mark the positions that are used for the computations according to α and γ: these computations both start in position $g(k'+n)$ and end in $f_{k'+n}(\alpha)$ and $f_{k'+n}(\gamma)$, respectively. Note that the set of marked positions never has a gap that exceeds k.

Now consider a computation that starts from instruction $f_{k'+n}(\beta)$ in $Y_{k'+n}$, a position in between $f_{k'+n}(\alpha)$ and $f_{k'+n}(\gamma)$. By Claim 1, the first n a-instructions have positions in between $f_{k'+n}(\alpha)$ and $f_{k'+n}(\gamma)$ and none of these are marked. Leaving out all marked positions and adjusting the associated jumps yields a piece of code, say Y, with smaller jumps, thus in C_{k-1}, that has the a-n-property. Because n was chosen arbitrarily, this contradicts the initial supposition that k was minimal. □

Theorem 1. For any $k \in \mathbb{N}^+$, not all threads in BTA can be expressed in C_k. This is also the case if thread extraction may start at arbitrary positions.

Proof. Fix some value k. Then, by Lemma 1 we can find a value n such that no $X \in C_k$ has the a-n-property. But we can define a finite thread that has this property. □

Bibliography

[BL00] Jan A. Bergstra and M. E. Loots, *Program Algebra for Component Code*, Formal Asp. Comput. **12** (2000), no. 1, 1–17.

[BL02] ———, *Program Algebra for Sequential Code*, J. Log. Algebr. Program. **51** (2002), no. 2, 125–156.

[BP09a] Jan A. Bergstra and Alban Ponse, *An Instruction Sequence Semigroup with Involutive Anti-Automorphisms*, Scientific Annals of Computer Science (2009), no. 19, 57–92.

[BP09b] ———, *An Instruction Sequence Semigroup with Involutive Anti-Automorphisms*, CoRR **abs/0903.1352v1** (2009).

[CP61] A. H. Clifford and G. B. Preston, *The Algebraic Theory of Semigroups, Volume I*, Mathematical Surveys, no. 7, American Mathematical Society, Providence, Rhode Island, 1961.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha, *The Java Language Specification*, third ed., Addison-Wesley, June 2005.

[ISO99] ISO, *ISO C Standard 1999*, Tech. report, 1999, ISO/IEC 9899:1999 draft.

[KR88] Brian W. Kernighan and Dennis Ritchie, *The C Programming Language*, second ed., Prentice-Hall, 1988.

[PvdZ06] Alban Ponse and Mark van der Zwaag, *An Introduction to Program and Thread Algebra*, CiE (Arnold Beckmann, Ulrich Berger, Benedikt Löwe, and John V. Tucker, eds.), Lecture Notes in Computer Science, vol. 3988, Springer, 2006, pp. 445–458.