The Böhm–Jacopini Theorem Is False, Propositionally

Dexter Kozen and Wei-Lung Dustin Tseng

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501, USA
{kozen,wdtseng}@cs.cornell.edu

Abstract. The Böhm–Jacopini theorem (Böhm and Jacopini, 1966) is a classical result of program schematology. It states that any deterministic flowchart program is equivalent to a while program. The theorem is usually formulated at the first-order interpreted or first-order uninterpreted (schematic) level, because the construction requires the introduction of auxiliary variables. Ashcroft and Manna (1972) and Kosaraju (1973) showed that this is unavoidable. As observed by a number of authors, a slightly more powerful structured programming construct, namely loop programs with multi-level breaks, is sufficient to represent all deterministic flowcharts without introducing auxiliary variables. Kosaraju (1973) established a strict hierarchy determined by the maximum depth of nesting allowed. In this paper we give a purely propositional account of these results. We reformulate the problems at the propositional level in terms of automata on guarded strings, the automata-theoretic counterpart to Kleene algebra with tests. Whereas the classical approaches do not distinguish between first-order and propositional levels of abstraction, we find that the purely propositional formulation allows a more streamlined mathematical treatment, using algebraic and topological concepts such as bisimulation and coinduction. Using these tools, we can give more mathematically rigorous formulations and simpler and more revealing proofs.

1 Introduction

Program schematology was one of the earliest topics in the mathematics of computing. A central problem that has been well studied over the years is that of transforming an unstructured flowgraph to structured form. A seminal result in this area is the Böhm–Jacopini theorem [2], which states that any deterministic flowchart program is equivalent to a while program. This classical theorem has reappeared in many contexts and has been reproved by many different methods. There are dozens of references on this topic; a few commonly cited ones are [1,8,9,10,11].

The Böhm–Jacopini theorem is usually formulated at the first-order interpreted or first-order uninterpreted (schematic) level, as was most early work in program schematology. The first-order formulation allows the introduction of

P. Audebaud and C. Paulin-Mohring (Eds.): MPC 2008, LNCS 5133, pp. 177–192, 2008.

auxiliary individual or Boolean variables to preserve information during the restructuring. This is an essential ingredient of the Böhm–Jacopini construction, and they asked whether it was strictly necessary. This question was answered affirmatively by Ashcroft and Manna [1] and Kosaraju [4].

Böhm and Jacopini's question, and Ashcroft and Manna and Kosaraju's solutions, were phrased in terms of the necessity of introducing auxiliary variables. This view is repeated in subsequent works, e.g. [8]. However, this is not really the best way to phrase the question. What was really shown was that a purely propositional formulation of the Böhm–Jacopini theorem is false: there is a deterministic propositional flowchart that is not equivalent to any propositional while program. This result is implicit in [1,4], although it was not stated this way.

As observed by a number of authors (e.g. [4,9]), a slightly more powerful structured programming construct, namely loops with multi-level breaks, is sufficient to represent all deterministic flowcharts without introducing auxiliary variables. Kosaraju [4] established a strict hierarchy based on the levels of the multi-level breaks that are allowed. He showed that for any $n \geq 1$, there exists a loop program with break m for $m \leq n$ that is not equivalent to any loop program with break m for $m \leq n-1$. Again, however, these results were formulated and proved at the first-order interpreted level, despite the fact that they are essentially propositional.

Inexpressibility proofs such as those of [1,4] that reason in terms of a particular first-order interpretation may appear contrived, because any number of other interpretations could serve the same purpose. One runs the risk of obscuring the underlying principles at work by the details of the particular construction, which are largely irrelevant. Moreover, the classical approach to program schematology relies heavily on graphs and combinatorial graph restructuring operations, which can be difficult to reason about formally.

Automata on guarded strings, the automata-theoretic counterpart of Kleene algebra with tests (KAT), provide an opportunity to reset the theory of program schemes on more rigorous algebraic foundations. We have found that a purely propositional, automata-theoretic reformulation of some of the questions mentioned above allows a more streamlined treatment. Algebraic and topological concepts such as bisimulation and coinduction, absent in earlier treatments, predominate here. We feel that the resulting proofs are simpler, more rigorous, and more revealing of the underlying principles at work.

This paper is organized as follows. In Section 2, we briefly discuss the differences between propositional and first-order formulations, and recall the basic definitions regarding guarded strings and automata with tests. We introduce a special restricted form of automata with tests, which we call *strictly deterministic*, corresponding to deterministic flowchart schemes. We argue that every deterministic flowchart scheme is semantically equivalent to a strictly deterministic automaton. Also in Section 2, we define bisimulation for strictly deterministic automata and mention several more or less standard results regarding bisimulations. We also recall the definition of the structured programming constructs

for while and loop programs and their semantics. Many proofs in this section are quite routine and are omitted.

Using these tools, we then give a purely propositional account of three known results: that the Böhm–Jacopini theorem is false at the propositional level, that loop programs with multi-level breaks are sufficient to represent all deterministic flowcharts, and that the Kosaraju hierarchy is strict. These results are proved in Sections 3, 4, and 5, respectively. We conclude with some open problems in Section 6.

2 Preliminaries

2.1 Propositional vs. First-Order Logic

The notions of functions on a domain and variables ranging over that domain are inherent in first-order logic, but are not present in propositional logic. Whereas we may consider a variable assignment $x := t$ as a primitive action in first-order program logic, a primitive action in propositional program logic is just a symbol. Since previous constructions establishing the Böhm–Jacopini theorem require the introduction of extra variables, they cannot be formalized at the propositional level of abstraction.

In this paper, we model propositional deterministic flowcharts and structured programs as strictly deterministic automata with tests, and we model program executions as guarded strings (both defined below). If desired, our propositional formulation can be extended with a first order interpretation. A subtle but important point is that all behaviors of a propositional program have a first-order realization; that is, given any guarded string representing a possible execution of a propositional program, there is a first-order interpretation that realizes that execution.

2.2 Guarded Strings

Guarded strings were introduced in [3]. They model program executions propositionally. Let Σ be a finite set of *action symbols* and T a finite set of *test symbols* disjoint from Σ. The symbols T generate a free Boolean algebra B; elements of B are called *tests*. An *atom* is a minimal nonzero element of B. The set of atoms is denoted At. The elements of At can be regarded either as conjunctions of literals of T (elements of T or their negations) or as truth assignments to T, thus $|\mathsf{At}| = 2^{|T|}$. We write $p, q, p_0, \ldots$ for elements of Σ and $\alpha, \beta, \alpha_0, \ldots$ for elements of At. A *guarded string* is a finite alternating sequence of atoms and actions, beginning and ending with an atom; that is, an element of $(\mathsf{At} \cdot \Sigma)^* \cdot \mathsf{At}$. In other words, guarded strings represent the join-irreducible elements of the free KAT on generators Σ and T. Intuitively, a guarded string records the sequence of primitive actions taken by a program and the tests that are true between any two successive primitive actions.

We will also consider infinite guarded strings, which are members of $(\mathsf{At} \cdot \Sigma)^\omega$, but will always qualify with the adjective "infinite" when doing so.

2.3 Automata with Tests

Automata with tests, also known as automata on guarded strings, were studied in [5]. They are the automata-theoretic counterpart to Kleene algebra with tests (KAT). In the formalism of [5], they have two types of transitions, *action transitions* and *test transitions*, and operate over guarded strings. An ordinary automaton with null transitions is just an automaton with tests over the two-element Boolean algebra. Many of the constructions of ordinary finite-state automata, such as determinization and state minimization, extend readily to automata with tests. In particular, there is a version of Kleene's theorem showing that these automata are equivalent in expressive power to expressions in the language of KAT. See [5] for a more detailed introduction.

2.4 Strictly Deterministic Automata

For the purposes of this paper, we will only need to consider a limited class of automata with tests corresponding to deterministic propositional flowchart schemes. Since actions are uniquely determined, we may elide the action states to obtain what we call a *strictly deterministic automaton*.

Intuitively, a strictly deterministic automaton operates by starting in its start state and scanning a sequence of atoms, which we can view as provided by an external agent. For each atom in succession, the automaton responds deterministically either by emitting an action symbol and moving to a new state, by halting, or by failing, according to its transition function.

Formally, a *strictly deterministic automaton* over Σ and T is a tuple

$$M = (Q, \delta, \mathsf{start}),$$

where Q is a (possibly infinite) set of *states*, $\mathsf{start} \in Q$ is the *start state*, and δ is a *transition function*

$$\delta : Q \times \mathsf{At} \to (\Sigma \times Q) + \{\mathsf{halt}, \mathsf{fail}\},$$

where $+$ denotes disjoint (marked) union. The elements halt and fail are not states, but universal constants used by an automaton to represent halting and failing, respectively. The components Q, δ, and start may be adorned with the subscript M where necessary to distinguish between automata. States are denoted by $s, t, u, v, \ldots$.

A *trace* in M is a finite or infinite alternating sequence of states and atoms specifying a path through M. Formally, a *trace* is a sequence σ in

$$(Q \cdot \mathsf{At})^* \cdot Q \;+\; (Q \cdot \mathsf{At})^\omega$$

such that for every substring of σ of the form $u\alpha v$, $\delta(u, \alpha) = (p, v)$ for some $p \in \Sigma$. The first state of σ is denoted $\mathsf{first}\,\sigma$ and the last state (if it exists) is denoted $\mathsf{last}\,\sigma$.

Given a state s and an infinite sequence of atoms σ, there is a unique finite or infinite trace $\mathsf{tr}(s, \sigma)$ determined intuitively by starting in state s and running

the automaton, making choices at each successive state according to the next atom in the sequence σ as determined by the transition function δ. The trace is finite iff M halts or fails along the way, even though σ is infinite. Formally, the map

$$\mathsf{tr} : Q \times \mathsf{At}^\omega \to (Q \cdot \mathsf{At})^* \cdot Q \ + \ (Q \cdot \mathsf{At})^\omega$$

is defined coinductively as follows:

$$\mathsf{tr}(s, \alpha\, \sigma) \stackrel{\text{def}}{=} \begin{cases} s \cdot \alpha \cdot \mathsf{tr}(t, \sigma) & \text{if } \delta(s, \alpha) = (p, t) \\ s & \text{if } \delta(s, \alpha) \in \{\mathsf{halt}, \mathsf{fail}\}. \end{cases}$$

This definition determines $\mathsf{tr}(s, \sigma)$ uniquely for all $s \in Q$ and $\sigma \in \mathsf{At}^\omega$.

A similar definition holds for guarded strings. Here we also allow infinite guarded strings as well as finite ones. Given a starting state s and an infinite sequence of atoms σ, there is at most one finite or infinite guarded string $\mathsf{gs}(s, \sigma)$ obtained by running the automaton starting in state s. Formally, the partial map

$$\mathsf{gs} : Q \times \mathsf{At}^\omega \to (\mathsf{At} \cdot \Sigma)^* \cdot \mathsf{At} \ + \ (\mathsf{At} \cdot \Sigma)^\omega$$

is defined coinductively as follows:

$$\mathsf{gs}(s, \alpha\, \sigma) \stackrel{\text{def}}{=} \begin{cases} \alpha \cdot p \cdot \mathsf{gs}(t, \sigma) & \text{if } \delta(s, \alpha) = (p, t) \\ \alpha & \text{if } \delta(s, \alpha) = \mathsf{halt} \\ \text{undefined} & \text{if } \delta(s, \alpha) = \mathsf{fail}. \end{cases}$$

As with traces, $\mathsf{gs}(s, \sigma)$ is uniquely determined for all $s \in Q$ and $\sigma \in A^\omega$.

The set of (finite) guarded strings represented by the automaton M is

$$\mathsf{GS}(M) \stackrel{\text{def}}{=} \{\mathsf{gs}(\mathsf{start}_M, \sigma) \mid \sigma \in \mathsf{At}^\omega\} \cap (\mathsf{At} \cdot \Sigma)^* \cdot \mathsf{At}.$$

Two automata are considered semantically equivalent if they represent the same set of finite guarded strings.

The transition function δ determines a map

$$\widehat{\delta} : Q \times \mathsf{At}^* \to Q + \{\mathsf{halt}, \mathsf{fail}\}$$

defined inductively as follows:

$$\widehat{\delta}(s, \sigma) \stackrel{\text{def}}{=} \begin{cases} s, & \text{if } \sigma = \varepsilon \\ \widehat{\delta}(t, \tau), & \text{if } \sigma = \alpha\tau \text{ and } \delta(s, \alpha) = (p, t) \\ \mathsf{halt}, & \text{if } \sigma = \alpha\tau \text{ and } \delta(s, \alpha) = \mathsf{halt} \\ \mathsf{fail}, & \text{if } \sigma = \alpha\tau \text{ and } \delta(s, \alpha) = \mathsf{fail}. \end{cases}$$

This is either the state that the machine is in after scanning σ starting in state s, or halt or fail if the machine halts or fails while scanning σ starting in state s.

2.5 Bisimulation

Let M and N be two strictly deterministic automata. A *bisimulation* between M and N is a binary relation $\equiv$ between Q_M and Q_N such that

(i) $\mathsf{start}_M \equiv \mathsf{start}_N$, and
(ii) if $s \in Q_M$, $t \in Q_N$, and $s \equiv t$, then for all $\alpha \in \mathsf{At}$,
  (a) $\delta_M(s,\alpha) = \mathsf{halt}$ iff $\delta_N(t,\alpha) = \mathsf{halt}$;
  (b) $\delta_M(s,\alpha) = \mathsf{fail}$ iff $\delta_N(t,\alpha) = \mathsf{fail}$; and
  (c) if $\delta_M(s,\alpha) = (p,s')$ and $\delta_N(t,\alpha) = (q,t')$, then $p = q$ and $s' \equiv t'$.

M and N are said to be *bisimilar* if there exists a bisimulation between M and N. An *autobisimulation* is a bisimulation between M and itself.

Bisimulations are closed under relational composition and arbitrary union, and the identity relation on an automaton is an autobisimulation. Thus the reflexive transitive closure of an autobisimulation is again an autobisimulation. Moreover, if two automata are bisimilar, then there is a unique maximum bisimulation between them, namely the union of all bisimulations between them. We provide three lemmas regarding bisimulation.

To show $\mathsf{GS}(M) = \mathsf{GS}(N)$, it suffices to show that M and N are bisimilar:

Lemma 1. *If M and N are bisimilar, then $\mathsf{GS}(M) = \mathsf{GS}(N)$.*

More interestingly, under certain mild conditions, the converse holds as well:

Lemma 2. *Suppose $\mathsf{GS}(M) = \mathsf{GS}(N)$, M does not contain a* fail *transition, and* halt *is accessible from every state of M that is accessible from start_M. Then M and N are bisimilar.*

Proof. For $s \in Q_M$ and $t \in Q_N$, set

$$s \equiv t \stackrel{\text{def}}{\iff} \forall \sigma \in \mathsf{At}^\omega \ \mathsf{gs}_M(s,\sigma) = \mathsf{gs}_N(t,\sigma).$$

We show that $\equiv$ is a bisimulation. If $s \equiv t$, then for all $\alpha \in \mathsf{At}$ and $\sigma \in \mathsf{At}^\omega$,

$$\delta_M(s,\alpha) = \mathsf{halt} \Leftrightarrow \mathsf{gs}_M(s,\alpha\sigma) = \alpha \ \Leftrightarrow \ \mathsf{gs}_N(t,\alpha\sigma) = \alpha \ \Leftrightarrow \ \delta_N(t,\alpha) = \mathsf{halt}$$

and similarly for fail, and if $\delta_M(s,\alpha) = (p,s')$ and $\delta_N(t,\alpha) = (q,t')$, then

$$\alpha \cdot p \cdot \mathsf{gs}_M(s',\sigma) = \mathsf{gs}_M(s,\alpha\sigma) \ = \ \mathsf{gs}_N(t,\alpha\sigma) \ = \ \alpha \cdot q \cdot \mathsf{gs}_N(t',\sigma),$$

thus $p = q$ and $\mathsf{gs}_M(s',\sigma) = \mathsf{gs}_N(t',\sigma)$. As σ was arbitrary, $s' \equiv t'$. This establishes property (ii) of bisimulation.

It remains to show property (i); that is, $\mathsf{start}_M \equiv \mathsf{start}_N$, or in other words, $\mathsf{gs}_M(\mathsf{start}_M,\sigma) = \mathsf{gs}_N(\mathsf{start}_N,\sigma)$ for all σ. By assumption, $\mathsf{GS}(M) = \mathsf{GS}(N)$, so if $\mathsf{gs}_M(\mathsf{start}_M,\sigma)$ is finite, then so is $\mathsf{gs}_N(\mathsf{start}_N,\sigma)$ and they are equal. Thus the functions

$$\mathsf{gs}_M(\mathsf{start}_M,-) : \mathsf{At}^\omega \to (\mathsf{At}\cdot\Sigma)^* \cdot \mathsf{At} + (\mathsf{At}\cdot\Sigma)^\omega$$
$$\mathsf{gs}_N(\mathsf{start}_N,-) : \mathsf{At}^\omega \to (\mathsf{At}\cdot\Sigma)^* \cdot \mathsf{At} + (\mathsf{At}\cdot\Sigma)^\omega$$

agree on the set $\{\sigma \mid \mathsf{gs}_M(\mathsf{start}_M, \sigma) \text{ is finite}\}$. The accessibility condition in the statement of the lemma implies that this set is dense in At^ω under the usual metric topology on ω-sequences[1]. Moreover, the two functions are continuous, and continuous functions that agree on a dense set must agree everywhere. □

Lemma 3. *If M and N are bisimilar under $\equiv$, then for any $\sigma \in \mathsf{At}^*$, either both $\widehat{\delta}_M(\mathsf{start}_M, \sigma)$ and $\widehat{\delta}_N(\mathsf{start}_N, \sigma)$ are states, both are* halt*, or both are* fail*; and if they are states, then they are related by $\equiv$.*

2.6 Structured Programming Constructs

Deterministic while programs are formed inductively from sequential composition $(p\,;q)$, conditional tests (if b then p else q), and while loops (while b do p), where b is a test and p, q are programs. We also include instructions skip (do nothing) and fail (looping or abnormal termination), although these constructs are redundant, being semantically equivalent to while false do p and while true do skip, respectively. We do not include a halt instruction; a program terminates normally by falling off the end.

Every while program can be converted to an equivalent strictly deterministic automaton. One first converts the program to a KAT term using the standard translation

$$p\,;q = pq \qquad \text{if } b \text{ then } p \text{ else } q = bp + \overline{b}q \qquad \text{while } b \text{ do } p = (bp)^*\overline{b},$$

then applies Kleene's theorem for KAT to yield an automaton with test and action states [5], which can be viewed as a deterministic flowchart F. One can then define a strictly deterministic transition function δ on the states of F as follows. For any state s and atom α, start at s and follow test transitions enabled by α until encountering an action state or a halt state. If an action state is encountered, let p be the label of the transition from that state and set $\delta(s, \alpha) = (p, t)$, where t is the target state of the transition. If a halt state is encountered, set $\delta(s, \alpha) = \mathsf{halt}$. If neither of these occur, that is, if the process traces a cycle of enabled test transitions, set $\delta(s, \alpha) = \mathsf{fail}$.

By restricting to the start state and the targets of action transitions, one obtains a strictly deterministic automaton of the form of Section 2.4. An example is shown in Fig. 1. In that figure, an edge from s to t labeled αp denotes the transition $\delta(s, \alpha) = (p, t)$. Note that the set of states of the strictly deterministic automaton is a subset of the states of the original automaton. The conversion of deterministic flowcharts to strictly deterministic automata does not change the set of guarded strings accepted. Moreover, by Kleene's theorem for KAT [5], this is the same as the set of guarded strings represented by the equivalent KAT expression.

In addition to the usual while program constructs, we consider the looping construct loop with nonlocal breaks break n, $n \geq 1$. After conversion to a deterministic flowchart, every loop instruction ℓ has one entry point entry_ℓ and

[1] The distance between two sequences is 2^{-n} if they agree on their first n symbols but differ on their $n+1$st symbol.

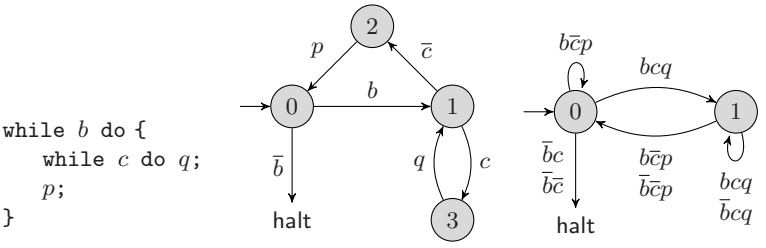

Fig. 1. A while program and its corresponding deterministic flowchart and strictly deterministic automaton

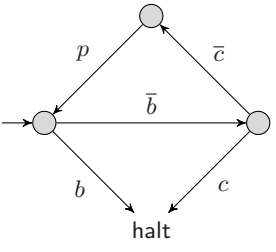

Fig. 2. An example from [4]

one exit point exit_ℓ. Intuitively, the instruction break n transfers control to exit_ℓ, where ℓ is the nth loop in whose scope the break n instruction occurs, counting from innermost to outermost. For while loops ℓ, $\mathsf{entry}_\ell = \mathsf{exit}_\ell$.

One can give a rigorous compositional semantics and an equational axiomatization of loop and break n, but this topic deserves a careful and systematic development that would be too much of a digression for the purposes of this paper, so we defer it to a forthcoming paper [6].

Allowing Boolean combinations of primitive tests in while loops and conditionals is quite natural and allows more flexibility than primitive tests alone. For instance, Kosaraju [4, Theorem 2] presents the flowchart of Fig. 2 as an example of a deterministic program that is not equivalent to any while program. This is true under his definition, but for the uninteresting reason that only primitive tests are allowed. Allowing Boolean combinations, the flowchart is equivalent to while $\overline{b}\overline{c}$ do p. The counterexample of Ashcroft and Manna [1] is much more complicated, requiring 13 nodes. Both proofs are rather lengthy and reason in terms of a particular first-order interpretation.

3 While Programs Are Not Sufficient

In this section we give a three-state strictly deterministic automaton M that cannot be represented by any while program. The states are $0, 1, 2$ with start

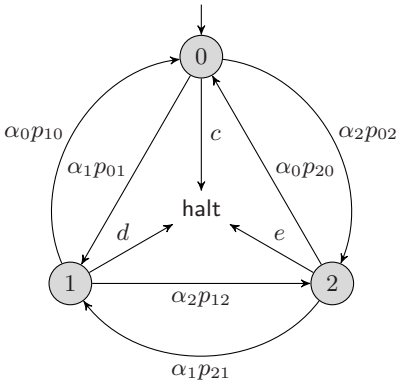

Fig. 3. A strictly deterministic automaton not equivalent to any while program

state 0. The primitive actions are p_{st} for $s, t \in \{0, 1, 2\}$, $s \neq t$, and the primitive tests are a, b, giving four atoms $\alpha_0, \ldots, \alpha_3$. The transitions are $\delta(s, \alpha_t) = (p_{st}, t)$ for $s, t \in \{0, 1, 2\}$, $s \neq t$, and $\delta(s, \alpha_3) = \delta(s, \alpha_s) = \mathsf{halt}$. The automaton M is illustrated in Fig. 3. For example, the edge from 0 to 2 labeled $\alpha_2 p_{02}$ represents the transition $\delta(0, \alpha_2) = (p_{02}, 2)$. The tests c, d, e represent $\alpha_0 + \alpha_3$, $\alpha_1 + \alpha_3$, and $\alpha_2 + \alpha_3$, respectively. The edge labeled c represents the two transitions $\delta(0, \alpha_0) = \delta(0, \alpha_3) = \mathsf{halt}$.

The automaton M has no nontrivial autobisimulation, since $\delta(s, \alpha_t) \neq \delta(t, \alpha_t)$ for $s \neq t$.

Theorem 1. *The strictly deterministic automaton M of Fig. 3 is not equivalent to any* while *program.*

Proof. Suppose for a contradiction that there exists a while program W equivalent to M; that is, such that $\mathsf{GS}(W) = \mathsf{GS}(M)$. Then W has a representation as a deterministic flowchart, and as a consequence of the construction of Section 2.6, as a strictly deterministic automaton S whose states are a subset of the states of W. We can assume without loss of generality that all states of W are accessible from start_W under a string in $\{\alpha_i \mid 0 \leq i \leq 2\}^*$; inaccessible states can be deleted with impunity. By Lemma 2, M and S are bisimilar.

For $s \in Q_S$, let $\mathsf{bisim}(s) \in Q_M$ be the unique state in M to which s is bisimilar. The state $\mathsf{bisim}(s)$ is unique, otherwise by transitivity there would be two bisimilar states of M, contradicting the fact that M is reduced. Also, since $\mathsf{start}_W \in Q_S$, $\mathsf{bisim}(\mathsf{start}_W)$ exists and is equal to 0.

Let $\ell = \mathsf{while}\ c\ \mathsf{do}\ r$ be a while loop in W of maximal depth, and let $s_0 = \mathsf{entry}_\ell = \mathsf{exit}_\ell$. Note that s_0 is not necessarily in Q_S. Let $s, t \in Q_S$ and $\alpha \in \mathsf{At}$ such that $\widehat{\delta}(s, \alpha) = (p_{ij}, t)$, s is not in the body of ℓ, and t is in the body of ℓ. It may be that $s = s_0$, but not necessarily. The states s and t exist, otherwise the body of ℓ is inaccessible. By symmetry, we may assume without loss of generality that $i = 0$ and $j = 1$. Thus $\mathsf{bisim}(s) = 0$, $\mathsf{bisim}(t) = 1$, $\alpha = \alpha_1$, and $\delta(s, \alpha_1) = \delta(s_0, \alpha_1) = (p_{01}, t)$.

Let σ be a maximum-length string of the form $(\alpha_2\alpha_1)^n$ or $(\alpha_2\alpha_1)^n\alpha_2$ such that the computation in W under σ starting from t does not meet s_0. The string σ exists, since ℓ has no inner loops, so all sufficiently long computations will loop back to s_0. Let $u = \widehat{\delta}(t, \sigma)$. The string σ cannot be of the form $(\alpha_2\alpha_1)^n\alpha_2$, because then we would have $\mathsf{bisim}(u) = 2$ and $\delta(u, \alpha_1) = \delta(s_0, \alpha_1) = (p_{21}, w)$ for some w, a contradiction. Thus σ is of the form $(\alpha_2\alpha_1)^n$, and $\delta(s_0, \alpha_2) = (p_{12}, w)$ for some w.

Suppose there is a state y in the body of ℓ with $\mathsf{bisim}(y) \in \{0, 2\}$. Consider a maximum-length string of alternating α_2 and α_0 such that the computation sequence under this string starting at y does not meet s_0. The first atom of the sequence is α_2 if $\mathsf{bisim}(y) = 0$ and α_0 if $\mathsf{bisim}(y) = 2$. As above, the last state v of the sequence cannot be bisimilar to 0, because then we would have $\delta(v, \alpha_2) = \delta(s_0, \alpha_2) = (p_{02}, z)$ for some z, a contradiction. Thus we must have $\delta(v, \alpha_0) = \delta(s_0, \alpha_0) = (p_{20}, x)$ for some x.

Collecting information about ℓ so far, we have

$$\delta(s_0, \alpha_0) = (p_{20}, x), \qquad \delta(s_0, \alpha_1) = (p_{01}, t), \qquad \delta(s_0, \alpha_2) = (p_{12}, w).$$

But now if we start from t and follow a sufficiently long path of the form $(\alpha_0\alpha_2\alpha_1)^*$, we will achieve a contradiction no matter what. Thus our assumption that the body of ℓ contains a state y with $\mathsf{bisim}(y) \in \{0, 2\}$ was fallacious. The body of ℓ contains only the state $t \in Q_S$ with $\mathsf{bisim}(t) = 1$, and x and w are outside the body of ℓ. The loop ℓ is only entered under α_1, after which it performs the action p_{01} and immediately halts or exits the loop. Thus ℓ is equivalent to a conditional test.

By inductively replacing all maximally deeply nested while loops with equivalent conditional tests in this way, we can eventually eliminate all while loops. This is a contradiction. □

Theorem 1 shows that the Böhm-Jacopini theorem is false propositionally. In Section 5, a similar argument is used to prove the Kosaraju hierarchy theorem [4].

4 Loop Programs with Multi-level Breaks

As we saw in Section 3, while programs cannot express all programs represented by strictly deterministic automata. On the other hand, if an automaton has no cycles, then by duplicating states it can be converted to a tree, which is equivalent to a program built from just the if-then-else construct.

Motivated by this idea, our construction will first construct an equivalent tree-like automaton consisting of (downward-directed) tree transitions and (upward-directed) back transitions, then convert the resulting tree-like automaton to a loop program. This is done in three steps. The first step "unwinds" the original automaton to an infinite tree. This is a fairly standard construction, although we do it here with traces and bisimulations. The second step identifies

states in the infinite tree with equivalent ancestors to obtain a finite tree-like automaton. In both steps, there is a bisimulation that guarantees equivalence. Finally, the tree-like automaton is converted to a loop program by using loop and break n to effect the back transitions and halting.

The "unwinding" of an automaton M to an infinite tree is done formally as follows. Let

$$U \stackrel{\text{def}}{=} (Q_U, \delta_U, \mathsf{start}_U)$$

where

$$\begin{aligned} Q_U &\stackrel{\text{def}}{=} \{\text{finite traces } \sigma \text{ of } M \text{ such that } \mathsf{first}\,\sigma = \mathsf{start}_M\}, \\ \delta_U(\sigma,\alpha) &\stackrel{\text{def}}{=} \begin{cases} (p,\, \sigma\alpha t) & \text{if } \delta_M(\mathsf{last}\,\sigma,\alpha) = (p,t) \\ \mathsf{halt} & \text{if } \delta_M(\mathsf{last}\,\sigma,\alpha) = \mathsf{halt} \\ \mathsf{fail} & \text{if } \delta_M(\mathsf{last}\,\sigma,\alpha) = \mathsf{fail}, \end{cases} \\ \mathsf{start}_U &\stackrel{\text{def}}{=} \mathsf{start}_M. \end{aligned}$$

Lemma 4. *The relation* $\{(\sigma, \mathsf{last}\,\sigma) \mid \sigma \in Q_U\}$ *is a bisimulation between* U *and* M*. Thus by Lemma 1,* $\mathsf{GS}(U) = \mathsf{GS}(M)$*.*

A *congruence* on M is an equivalence relation $\equiv$ that is an autobisimulation on M. Property (ii) of bisimulations says that the action of δ is well defined on $\equiv$-congruence classes, thus we can form the quotient automaton $M/\!\equiv$ whose states are the $\equiv$-congruence classes. Denote the congruence class of state u by $[u]$.

Lemma 5. *The relation* $\{(u, [u]) \mid u \in Q_M\}$ *is a bisimulation between* M *and* $M/\!\equiv$*. Thus by Lemma 1,* $\mathsf{GS}(M) = \mathsf{GS}(M/\!\equiv)$*.*

We can now use this construction to form a tree-like automaton with finitely many states equivalent to U. Here *tree-like* means that it has tree edges that form a rooted tree, but also may contain back edges to ancestors.

Recall that the states of U are the finite traces of M starting with start_M. For $\sigma, \tau \in Q_U$, set $\sigma\, R\, \tau$ iff

- all states of τ occur exactly once in τ except $\mathsf{last}\,\tau$, which occurs exactly twice;
- σ is the unique proper prefix of τ such that $\mathsf{last}\,\sigma = \mathsf{last}\,\tau$.

Let $\equiv$ be the smallest congruence containing R. That is, $\equiv$ is the smallest binary relation on Q_U such that

(i) $\equiv$ contains R,
(ii) $\equiv$ is an equivalence relation, and
(iii) if $\sigma \equiv \tau$ and $\delta_M(\mathsf{last}\,\sigma, \alpha) = \delta_M(\mathsf{last}\,\tau, \alpha) = (p, v)$, then $\sigma\alpha v \equiv \tau\alpha v$.

It can be shown inductively that $\mathsf{last}\,\sigma = \mathsf{last}\,\tau$ whenever $\sigma \equiv \tau$, so condition (iii) makes sense. The quotient automaton $U/\!\equiv$ has finitely many states, at most $(|Q_M| - 1)!$ in fact, since each $\equiv$-congruence class contains a unique trace

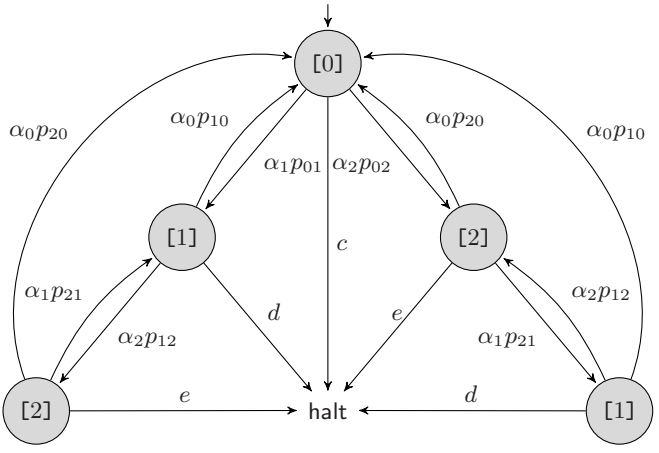

Fig. 4. A tree-like automaton equivalent to the automaton of Fig. 3

with no repeated states beginning with start_M. This trace is of minimal length among all elements of its $\equiv$-class. It can be obtained from any other element of the class by repeatedly deleting the subtrace between the first recurring state and its earlier occurrence.

We can view the states of $U/\equiv$ as arranged in a tree with root $[\mathsf{start}_M]$, tree edges to descendants, and back edges to ancestors. By Lemma 5, $\mathsf{GS}(U) = \mathsf{GS}(U/\equiv)$.

Now we can convert this automaton to a loop program as follows. Let C be the set of traces of M with no repeated states beginning with start_M. This is the set of canonical representatives of the $\equiv$-classes. Let $\alpha_1, \ldots, \alpha_m$ be the elements of At. For each $\sigma \in C$, let L_σ be the following loop program:

```
loop {
   if α1 then S1
   else if α2 then S2
   ...
   else if αm then Sm
}
```

where

(i) if $\delta_M(\mathsf{last}\,\sigma, \alpha_i) = (p, t)$, then

$$S_i = \begin{cases} p\ ;\ L_{\sigma\alpha_i t} & \text{if } t \text{ does not occur in } \sigma, \\ p & \text{if } t = \mathsf{last}\,\sigma, \\ p\ ;\ \mathsf{break}\ n & \text{if } t \text{ occurs in } \sigma \text{ but } t \neq \mathsf{last}\,\sigma, \end{cases}$$

where in the last case, n is the number of states occurring after t in σ;

(ii) if $\delta_M(\mathsf{last}\,\sigma, \alpha_i) = \mathsf{halt}$, then $S_i = \mathsf{break}\ n$, where n is the number of states in σ; and
(iii) if $\delta_M(\mathsf{last}\,\sigma, \alpha_i) = \mathsf{fail}$, then $S_i = \mathsf{loop\ skip}$.

The choice of n in the last case of (i) causes control to return to the top of L_τ, where τ is the unique prefix of σ such that $\mathsf{last}\,\tau = t$. This is tantamount to taking the back edge from the node of the tree represented by σ to its ancestor represented by τ. The choice of n in case (ii) causes the program to halt by exiting the outermost loop L_{start}.

Example 1. Fig. 4 shows a tree-like automaton equivalent to the automaton of Fig. 3. (The central edges leading to halt are not considered part of the tree, since halt is not a state of the automaton.) A corresponding loop program is shown in Fig. 5. This is not exactly the program that would be produced by the construction given above; we have removed the innermost loops to save space.

```
loop {
   if ā then break 1;
   if b then {
      p;
      loop {
         if ā then break 2;
         if b̄ then { t; break 1; }
         else {
            s;
            if ā then break 2;
            if b then { v; break 1; }
            else w;
         }
      }
   } else {
      q;
      loop {
         if ā then break 2;
         if b then { v; break 1; }
         else {
            w;
            if ā then break 2;
            if b̄ then { t; break 1; }
            else s;
         }
      }
   }
}
```

Fig. 5. A loop program equivalent to the automaton of Fig. 4; we have removed the innermost loops to save space

5 The Loop Hierarchy

Now we give an alternative proof of the hierarchy result of Kosaraju [4], namely that there is a strict hierarchy of loop programs determined by the depth of nesting of loop instructions.

We construct an automaton P_n as follows. The states of P_n are all strings over the alphabet $\{0, 1, \ldots, n-1\}$ with no repeated letters, including the empty string ε. There are roughly $n!e$ states. The atoms and actions are $0, 1, \ldots, n-1$ and E. The transition i appends i to the current string if it does not already occur, or else truncates back to the prefix ending in i if it does occur. The transition E erases the string. We also include an atom H such that $\delta(s, H) = \mathsf{halt}$ for states s of maximum length n and $\delta(s, H) = \mathsf{fail}$ for the other states. This precludes nontrivial autobisimulations.

An illustration of a depth-4 implementation of P_7 is shown in Fig. 6. Each box represents a loop instruction. Only four paths of the nested loop program are shown; there are many others not shown. There is one top-level loop, $\binom{n}{2}2!$ second-level loops, $\binom{n}{4}4!$ third-level loops within each second-level loop, etc.

At the entry point of each loop, there is a multiway branch depending on the current atom. The resulting action is the same as the atom, and the new state is the one whose last symbol is the action just performed (except for ε, which is only obtained by E). Note that every prefix of every string occurs in the same loop or an ancestor, therefore is accessible by a break instruction, and every string obtained by appending one symbol is in the same loop or a child loop.

For example, suppose the current state is 012. If we perform action 3, we would enter the subloop below 012 containing 0123. If we perform action 4, we would enter a parallel subloop not shown. If we perform action 1, we would loop to the top of the current loop, execute the action 1, and enter state 01.

Theorem 2. *The program P_n can be implemented in depth $\lfloor n/2+1 \rfloor$ and no less.*

Proof. For the upper bound, Fig. 6 illustrates the pattern that achieves $\lfloor n/2+1 \rfloor$.

For the lower bound, the proof is by induction on n. The idea is illustrated in Fig. 6: note that all strings in the 01 subloop begin with 01, so it has the same structure as the outer loop, but with two fewer letters. We actually prove a stronger result, namely that the bound holds irrespective of which state is the start state.

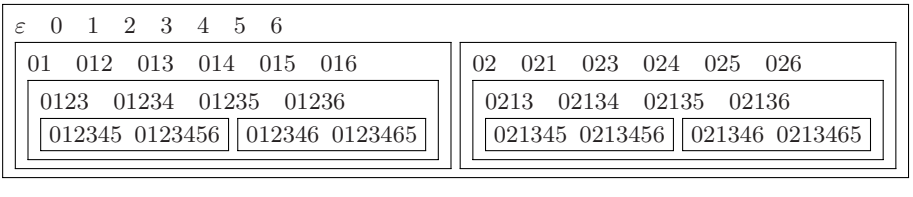

Fig. 6. Automaton P_7 implemented with a loop program of depth 4

The basis for $n = 0$ and $n = 1$ is trivial, since there always must be at least one loop. For $n = 2$, there are only three transitions but five states requiring self-loops, so the depth must be at least two.

Now let $n \geq 3$ and let W be any implementation of P_n. Let ℓ_0 be an outer loop of W and $s_0 = \mathsf{entry}_{\ell_0}$. There must be some pair i, j such that ij does not appear as a prefix of any x for $\delta(s_0, k) = (k, x)$. Say $ij = 01$ without loss of generality.

Consider the subprogram ℓ_1 consisting of all states of W that are accessible from 01 after deleting the transitions E and 0 (that is, setting them to fail). Since all states of ℓ_1 have prefix 01, the entry point s_0 of ℓ_0 is no longer accessible, so the outer loop can be deleted. The represented automaton is isomorphic to P_{n-2}. The transition 1 plays the role of E. By the induction hypothesis, it must have depth at least $\lfloor (n-2)/2 + 1 \rfloor$, thus ℓ_0 must have depth at least $\lfloor (n-2)/2 + 1 \rfloor + 1 = \lfloor n/2 + 1 \rfloor$.

6 Conclusion and Open Problems

We have shown three results giving upper and lower bounds on the power of various programming constructs to represent flowchart programs, modeled as automata on guarded strings. On the one hand, the simple three-state automaton in Section 3 cannot be represented by any while program. On the other hand, we present a congruence in Section 4 that transforms any automaton into a tree-like structure and show how a tree-like automaton can be turned into a loop program with multi-level breaks. We also give an alternative proof of Kosaraju's hierarchy result for loop programs with multi-level breaks.

We did not give a formal proof of equivalence between the tree-like automaton and its corresponding loop program with multi-level breaks constructed in Section 4. However, it is possible to prove their equivalence formally. The $\mathsf{break}\ n$ construct, and more generally the goto construct, although representing nonlocal flow of control, can nevertheless be given a formal equational semantics in the style of KAT. We have developed this semantics and an equational axiomatization and have shown how to use it to give rigorous proofs of the correctness of transformations like those of Section 4 [6].

One popular line of research has been to develop restructuring techniques that minimize the amount of duplication of code [8,10]. The construction given in Section 4 is as bad in this regard as it can possibly be: it transforms an n-state automaton to an $(n-1)!$-state tree-like automaton in the worst case. Are there more efficient transformations at the propositional level? Or is this an inescapable feature of the propositional formulation?

Acknowledgements

We would like to thank the reviewers for their helpful suggestions for improving the presentation. This work was supported by NSF grant CCF-0635028 and a NSF Graduate Research Fellowship.

References

1. Ashcroft, E., Manna, Z.: The translation of goto programs into while programs. In: Freiman, C.V., Griffith, J.E., Rosenfeld, J.L. (eds.) Proceedings of IFIP Congress 71, vol. 1, pp. 250–255. North-Holland, Amsterdam (1972)
2. Böhm, C., Jacopini, G.: Flow diagrams, Turing machines and languages with only two formation rules. In: Communications of the ACM, pp. 366–371 (May 1966)
3. Kaplan, D.M.: Regular expressions and the equivalence of programs. J. Comput. Syst. Sci. 3, 361–386 (1969)
4. Kosaraju, S.R.: Analysis of structured programs. In: Proc. 5th ACM Symp. Theory of Computing (STOC 1973), pp. 240–252. ACM, New York (1973)
5. Kozen, D.: Automata on guarded strings and applications. Matématica Contemporânea 24, 117–139 (2003)
6. Kozen, D.: Nonlocal flow of control and Kleene algebra with tests. Technical Report http://hdl.handle.net/1813/10595 Computing and Information Science, Cornell University (April 2008); Proc. 23rd IEEE Symp. Logic in Computer Science (LICS 2008) (to appear, June 2008)
7. Morris, P.H., Gray, R.A., Filman, R.E.: Goto removal based on regular expressions. J. Software Maintenance: Research and Practice 9(1), 47–66 (1997)
8. Oulsnam, G.: Unraveling unstructured programs. The Computer Journal 25(3), 379–387 (1982)
9. Peterson, W., Kasami, T., Tokura, N.: On the capabilities of while, repeat, and exit statements. Comm. Assoc. Comput. Mach. 16(8), 503–512 (1973)
10. Ramshaw, L.: Eliminating goto's while preserving program structure. Journal of the ACM 35(4), 893–920 (1988)
11. Williams, M., Ossher, H.: Conversion of unstructured flow diagrams into structured form. The Computer Journal 21(2), 161–167 (1978)