Bachelor thesis
Computer Science

Radboud University

Proving a folk theorem using Kleene Algebra with Tests

Author:
Rodin Aarssen
aarssen@science.ru.nl

Supervisor/first assessor:
prof. dr. J.H. Geuvers
herman@cs.ru.nl

Second assessor:
dr. A. Silva
alexandra@cs.ru.nl

July 8, 2014

Abstract

In this thesis, Kleene Algebra with Tests will be used to reason about program equivalence, as proposed by Kozen. We will go in-depth on the program transformations and will propose a way to handle the assignment rule.

Contents

1 Introduction **2**

2 Kleene Algebra with Tests **3**
2.1 Kleene Algebra 3
2.2 Examples of Kleene algebras 4
2.3 Kleene algebra with tests 5

3 The `WHILE` language **7**
3.1 Hoare logic 8
3.2 Assignment rule 11
3.3 Composition rule 11
3.4 Conditional rule 12
3.5 While rule 13
3.6 Weakening rule 13

4 A Folk Theorem **14**
4.1 Normalizing While programs 14
4.2 Conditional program 14
4.3 Program with nested while loops 15
4.4 Getting rid of postcomputations 16
4.5 Composition of programs 17

5 Conclusion **18**

Chapter 1

Introduction

A Kleene algebra is an algebraic structure that has many diverse applications, ranging from dynamic logic to language theory. A KA has the operators $+, \cdot, ^*, 0$ and 1 that satisfy a set of axioms.

In [6] and [7], Kozen introduces Kleene Algebra with Tests (KAT), an extended version of Kleene Algebra, and a transformation from the imperative programming language While to KAT. By doing this, he is able to reason about program equivalence. In particular, he wants to prove the equivalence of two While programs by proving that their translations to KAT are equal, reasoning in KAT.

In this thesis, we want to clarify what's been said by Kozen about program tranformations and equivalence. Furthermore, since KAT is purely propositional, it can not deal with assignment. Whereas Kozen does not consider the assignment rule for his system, we propose to use new KAT constants in order to be able to use assignment.

In the last Chapter of this thesis, a folk theorem on While programs will be covered. The theorem says that each While program is equivalent to another While program that contains at most *one* while-loop. By recursively applying program transformations (which are proven correct), this theorem will be proven.

Chapter 2

Kleene Algebra with Tests

2.1 Kleene Algebra

The notion of Kleene algebra ocurs in the literature at various places (e.g. [1], [7], [9]), where it is defined in non-equivalent formulations. In this thesis, Kozen's definition from [7] is used:

Definition 2.1 A Kleene Algebra is an algebraic structure $(\mathcal{K}, +, \cdot, ^*, 0, 1)$, which is a semiring with idempotent addition that also satisfies

$$1 + pp^* = p^* \tag{2.1}$$

$$1 + p^*p = p^* \tag{2.2}$$

$$q + pr \leq r \rightarrow p^*q \leq r \tag{2.3}$$

$$q + rp \leq r \rightarrow qp^* \leq r \tag{2.4}$$

where $\leq$ refers to the natural partial order on $\mathcal{K}$:

$$p \leq q \leftrightarrow p + q = q \tag{2.5}$$

Definition 2.2 A semiring is an algebraic structure that satisfies

$$p + (q + r) = (p + q) + r \tag{2.6}$$

$$p + q = q + p \tag{2.7}$$

$$p + 0 = p \tag{2.8}$$

$$p(qr) = (pq)r \tag{2.9}$$

$$1p = p \tag{2.10}$$

$$p1 = p \tag{2.11}$$

$$p(q + r) = pq + pr \tag{2.12}$$

$$(p + q)r = pq + qr \tag{2.13}$$

$$0p = 0 \tag{2.14}$$

$$p0 = 0 \tag{2.15}$$

Definition 2.3 A semiring with idempotent addition is a semiring that also satisfies

$$p + p = p \tag{2.16}$$

2.2 Examples of Kleene algebras

Kleene algebra generalizes the familiar notions from regular expressions. We will now give two examples of applicability of Kleene algebra.

Example 2.4 We can raise the theory of regular expressions to the level of Kleene algebra. Let e be a regular expression over the alphabet Σ. Then $\mathcal{L}(e) \subseteq \Sigma^*$, $\mathcal{L}(0) = \emptyset$ (the empty language), and $\mathcal{L}(1) = \{\lambda\}$ (the language with only the empty word). We translate the Equations 2.1-2.16, and the following equations all hold:

$$\mathcal{L}(1 + pp^*) = \mathcal{L}(p^*) \tag{2.17}$$
$$\mathcal{L}(1 + p^*p) = \mathcal{L}(p^*) \tag{2.18}$$
$$\mathcal{L}(q + pr) \leq \mathcal{L}(r) \rightarrow \mathcal{L}(p^*q) \leq \mathcal{L}(r) \tag{2.19}$$
$$\mathcal{L}(q + rp) \leq \mathcal{L}(r) \rightarrow \mathcal{L}(qp^*) \leq \mathcal{L}(r) \tag{2.20}$$
$$\mathcal{L}(p) \leq \mathcal{L}(q) \leftrightarrow \mathcal{L}(p + q) = \mathcal{L}(q) \tag{2.21}$$
$$\mathcal{L}(p + (q + r)) = \mathcal{L}((p + q) + r) \tag{2.22}$$
$$\mathcal{L}(p + q) = \mathcal{L}(q + p) \tag{2.23}$$
$$\mathcal{L}(p + 0) = \mathcal{L}(p) \tag{2.24}$$
$$\mathcal{L}(p(qr)) = \mathcal{L}((pq)r) \tag{2.25}$$
$$\mathcal{L}(1p) = \mathcal{L}(p) \tag{2.26}$$
$$\mathcal{L}(p1) = \mathcal{L}(p) \tag{2.27}$$
$$\mathcal{L}(p(q + r)) = \mathcal{L}(pq + pr) \tag{2.28}$$
$$\mathcal{L}((p + q)r) = \mathcal{L}(pq + qr) \tag{2.29}$$
$$\mathcal{L}(0p) = \mathcal{L}(0) \tag{2.30}$$
$$\mathcal{L}(p0) = \mathcal{L}(0) \tag{2.31}$$
$$\mathcal{L}(p + p) = \mathcal{L}(p) \tag{2.32}$$

Example 2.5 Another example of Kleene algebra is the theory of relational algebra. We look at P, Q, R as subsets of $A \times A$, 1 as the identity relation, 0 as the empty relation, $P + Q$ as the union $P \cup Q$, and $aPQb$ as $\exists c(aPc \wedge cQb)$, and aR^*b as aR^nb for a certain $n \geq 0$, where $R^0 = 1$ and $aR^nb = aRR^{n-1}b$ for $n > 1$. We translate the Equations 2.1-2.16 to the domain of relational algebra, and the following equations all hold in relational algebra:

$$1 + PP^* = P^* \tag{2.33}$$
$$1 + P^*P = P^* \tag{2.34}$$
$$Q + PR \leq R \to P^*Q \leq R \tag{2.35}$$
$$Q + RP \leq R \to QP^* \leq R \tag{2.36}$$
$$P \leq Q \leftrightarrow P + Q = Q \tag{2.37}$$
$$P + (Q + R) = (P + Q) + R \tag{2.38}$$
$$P + Q = Q + P \tag{2.39}$$
$$P + 0 = P \tag{2.40}$$
$$P(QR) = (PQ)R \tag{2.41}$$
$$1P = P \tag{2.42}$$
$$P1 = P \tag{2.43}$$
$$P(Q + R) = PQ + PR \tag{2.44}$$
$$(P + Q)R = PQ + QR \tag{2.45}$$
$$0P = 0 \tag{2.46}$$
$$P0 = 0 \tag{2.47}$$
$$P + P = P \tag{2.48}$$

2.3 Kleene algebra with tests

To give a precise definition of Kleene algebra with tests, we first need to define Boolean algebra in a few steps. The following definitions are adapted from [2].

Definition 2.6 A lattice is an algebraic structure $(P, \vee, \wedge)$. P is a non-empty set. $x \vee y$ is also referred to as x **join** y and is defined as the supremum (or least upper bound) of $\{x, y\}$. $x \wedge y$ is also referred to as x **meet** y and is defined as the infinum (or greatest lower bound) of $\{x, y\}$. It holds that for all $x, y \in P$, $x \vee y$ and $x \wedge y$ exist.

Theorem 2.7 If L is a lattice, then $\wedge$ and $\vee$ satisfy, for all $a, b, c \in L$:

$$(a \vee b) \vee c = a \vee (b \vee c) \qquad \text{associativity} \tag{2.49}$$
$$(a \wedge b) \wedge c = a \wedge (b \wedge c) \tag{2.50}$$
$$a \vee b = b \vee a \qquad \text{commutativity} \tag{2.51}$$
$$a \wedge b = b \wedge a \tag{2.52}$$
$$a \vee a = a \qquad \text{idempotency} \tag{2.53}$$
$$a \wedge a = a \tag{2.54}$$
$$a \vee (a \wedge b) = a \qquad \text{absorbtion} \tag{2.55}$$
$$a \wedge (a \vee b) = a \tag{2.56}$$

Proof. For a proof of this theorem, see [2]. □

Definition 2.8 A lattice L is said to have a **unit** or **identity** if there exists an element $1 \in L$, so that $a \wedge 1 = a$ for all $a \in L$. A lattice L is said to have a **zero** element if it has an element $0 \in L$, so that $a \vee 0 = a$ for all $a \in L$.

Definition 2.9 A distributive lattice is a lattice L that satisfies the distributive law:

$$\forall a, b, c \in L, (a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)) \tag{2.57}$$

Definition 2.10 A Boolean lattice is a distributive lattice L that also satisfies:

- L has 0, satisfying $0 \vee a = a$
- L has 1, satisfying $1 \wedge a = a$
- for each $a \in L$, there exists a unique complement $\overline{a} \in L$.
- $a \vee \overline{a} = 1$
- $a \wedge \overline{a} = 0$

Lemma 2.11 Let L be a boolean lattice. Then, the following equations hold:

- $\overline{0} = 1$ and $\overline{1} = 0$
- $\overline{\overline{0}} = a$ for all $a \in L$
- for all $a, b \in L$, $\overline{(a \vee b)} = \overline{a} \wedge \overline{b}$
- for all $a, b \in L$, $\overline{(a \wedge b)} = \overline{a} \vee \overline{b}$
- for all $a, b \in L$, $a \wedge b = \overline{(\overline{a} \vee \overline{b})}$

Proof. For a proof of this theorem, see [2]. □

Definition 2.12 A Boolean algebra is a special kind of boolean lattice for which the algebraic properties of $\wedge$, $\vee$ and the complementary operator are regarded as an integral part of the structure, with their properties being embodied in axioms. A Boolean algebra is then defined as an algebraic structure $(B, \vee, \wedge, \bar{}, 0, 1)$, so that

- $(B, \vee, \wedge)$ is a boolean lattice
- $a \vee 0 = a$ and $a \wedge 1 = a$ for all $a \in B$
- $a \vee \overline{a} = 1$ and $a \wedge \overline{a} = 0$ for all $a \in B$

From [7], we take the definition of a Kleene algebra with tests:

Definition 2.13 A Kleene algebra with tests is an algebraic structure $(\mathcal{K}, \mathcal{B}, +, \cdot, {}^*, \bar{}, 0, 1)$, where $(\mathcal{K}, +, \cdot, {}^*, 0, 1)$ is a Kleene algebra and $(\mathcal{B}, +, \cdot, \bar{}, 0, 1)$ a Boolean algebra. Furthermore, $\mathcal{B} \subseteq \mathcal{K}$.

Elements of $\mathcal{B}$ are called tests. In this thesis, p, q, r, s represent elements of $\mathcal{K}$, whereas a, b, c, d represent elements of $\mathcal{B}$.

Chapter 3

The WHILE language

In [8], Nielson and Nielson give a definition of a simple imperative programming language called WHILE.

Definition 3.1 The structure of while-constructs is:

$$a \quad ::= \quad n \mid x \mid a_1 + a_2 \mid a_1 \star a_2 \mid a_1 - a_2$$

$$b \quad ::= \quad \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2$$

$$S \quad ::= \quad x := a \mid \texttt{skip} \mid S_1; S_2 \mid \texttt{if } b \texttt{ then } p \texttt{ else } q \mid \texttt{while } b \texttt{ do } p$$

where a are the arithmetic expressions (**Num**), b the boolean expressions (**BExp**), S the statements (**Stm**), n the numerals (**Num**), and x the variables (**Var**).

For the rest of the theory, it's required that a boolean truth value (`true` or `false`) can be assigned to a variable. Therefore, we introduce yet another meta-variable β, that ranges over boolean variables (**BVar**). To deal with semantics, we change the **State**-function to $s := (s_a, s_b)$, where $s_a : \mathbf{Var} \to \mathbb{Z}$ is the state function as defined in [8], and $s_b : \mathbf{BVar} \to \{\texttt{true, false}\}$ is a function that adds `true` or `false` to an element β of **BVar** according to value of β in state s.

In [6], Kozen defines the normal form of a while program:

Definition 3.2 A **while** program is in normal form if it is in the form

p ; **while** b **do** q

where p and q do not contain a while-loop.

Since KAT is purely propositional, there is no domain of computation, analoguous to Propositional Dynamic Logic (PDL) ([4]). In [3], it is said that "Assignment is a non-propositional inference rule that deals with the internal structure of states. It is therefore disregarded in the embedding." We do want to be able to reason about program correctness (and thus about assignment) and therefore, we see statements involving state-dependent values as constants or KAT. This will be made more clear in the following.

We now introduce the transformation function $[\,]$: **while** $\to$ **KAT**, a function that transforms a while-statement to its equivalent in KAT, based on [6].

Definition 3.3 The function $[\,]$: **while** $\to$ **KAT** transforms a while-statement to a

KAT-expression. The boolean expressions are defined as

$$[\texttt{true}] = 1 \tag{3.1}$$
$$[\texttt{false}] = 0 \tag{3.2}$$
$$[a_1 = a_2] = \texttt{eq}(a_1, a_2) \tag{3.3}$$
$$[a_1 \leq a_2] = \texttt{leq}(a_1, a_2) \tag{3.4}$$
$$[\neg b] = \overline{b} \tag{3.5}$$
$$[b_1 \wedge b_2] = b_1 b_2 \tag{3.6}$$

The statements are defined as

$$[x := a] = \texttt{a}(x, a) \tag{3.7}$$
$$[\texttt{skip}] = 1 \tag{3.8}$$
$$[S_1; S_2] = S_1 S_2 \tag{3.9}$$
$$[\texttt{if}\ b\ \texttt{then}\ p\ \texttt{else}\ q] = bp + \overline{b}q \tag{3.10}$$
$$[\texttt{while}\ b\ \texttt{do}\ p] = (bp)^*\overline{b} \tag{3.11}$$

Here, the constant $\texttt{eq}(x, y)$ is a constant representing $x = y$, $\texttt{leq}(x, y)$ is a constant representing $x \leq y$ and $\texttt{a}(x, a)$ is a constant for the assignment $x := a$. For readability, we will sometimes write these KAT-constants as $[x = y]$, $[x \leq y]$ and $[x := a]$. Note that there exists no translation for the arithmetic expressions, since these values appear only in the KAT constants as indices. Furthermore, although the reader knows that, for instance, the constant $\texttt{eq}((x + 1) + 1, N)$ and $\texttt{eq}(x + 2, N)$ might represent the same truth value, this can not be inferred in KAT and they are in fact different constants.

3.1 Hoare logic

Hoare logic (introduced in 1969 by Hoare in [5]) is a formal system that can be used to reason about program correctness. A partial correctness assumption (PCA) is the basic notion of reasoning with Hoare logic. It is of the form $\{b\}p\{c\}$.

Definition 3.4 A PCA is a statement of the form

$$\{b\}p\{c\} \tag{3.12}$$

where p is a program and b and c are formulas.

Statement 3.12 intends to express that if b holds before p is executed and if p terminates, then c holds after termination. eq:leqbla The PCA in Statement 3.12 is encoded in KAT by either of the following equations:

$$bp\bar{c} = 0 \tag{3.13}$$
$$bp = bpc \tag{3.14}$$

Theorem 3.5 Statements 3.13 and 3.14 are equivalent in KAT.

Proof. From [7]: assuming 3.13, it's easily seen that

$$\begin{aligned} bp &= bp(c + \overline{c}) \\ &= bpc + bp\overline{c} \\ &= bpc \end{aligned}$$

Assuming 3.14,

$$\begin{aligned} bp\overline{c} &= bpc\overline{c} \\ &= bp0 \\ &= 0 \end{aligned}$$

□

We also define the translation function from Definition 3.3 for Hoare triples. We do that as follows:

$$[\{b\}p\{c\}] = (bp = bpc) \tag{3.15}$$

Example 3.6 Consider the following program:

x:=x+1; x:=x+2

The desired behavior of this program is that the value of x is incremented by 3. Therefore, we can write the Hoare triple

$$\{x = N\}x := x + 1; x := x + 2\{x = N + 3\} \tag{3.16}$$

Now, we make a derivation tree in Hoare logic:

$$\dfrac{\dfrac{x = N \rightarrow x + 1 = N + 1 \qquad \dfrac{}{\{x + 1 = N + 1\}x := x + 1\{x = N + 1\}}[\text{ass}]}{\{x = N\}x := x + 1\{x = N + 1\}}[\text{weak}] \qquad \dfrac{x = N + 1 \rightarrow x + 2 = N + 3 \qquad \dfrac{}{\{x + 2 = N + 3\}x := x + 2\{x = N + 3\}}[\text{ass}]}{\{x = N + 1\}x := x + 2\{x = N + 3\}}[\text{weak}]}{\{x = N\}x := x + 1; x := x + 2\{x = N + 3\}}[\text{comp}] \tag{3.17}$$

Now, we can say that from all the assignments at the top and all implications introduced by the weakening rule, it holds in Hoare logic that $\vdash \{x = N\}x := x + 1; x := x + 2\{x = N + 3\}$.

Specifically:

$$\begin{aligned}\{x+1=N+1\}x:=x+1\{x=N+1\}\wedge\\ \{x+2=N+3\}x:=x+2\{x=N+3\}\wedge\\ (x=N\to x+1=N+1)\wedge\\ (x=N+1\to x+2=N+3)\to\\ \{x=N\}x:=x+1;x:=x+2\{x=N+3\}\end{aligned}$$

We now translate this to KAT (renaming all the constants for readability) and see that this holds in KAT:

$$(e_1a_1=a_1a_1e_2)\wedge \tag{3.18}$$
$$(e_2a_2=e_2a_2e_4)\wedge \tag{3.19}$$
$$(e_5\le e_1)\wedge \tag{3.20}$$
$$(e_2\le e_3)\to \tag{3.21}$$
$$e_5a_1a_2=e_5a_1a_2e_4 \tag{3.22}$$

In general, a deduction tree of $\{b\}p\{c\}$ in Hoare logic yields a number of assignments at the top (which are axioms in Hoare logic), say Ass_1 to Ass_n, and a number of implications introduced by the weakening rule (which are all logically inductable), say ϕ_1 to ϕ_n. Then, it holds in KAT that

$$[Ass_1],\cdots,[Ass_n],[\phi_1],\cdots,[\phi_n]\vdash[\{b\}p\{c\}] \tag{3.23}$$

Note that $[p\to q]=p\le q$.

In the following sections, the Hoare inference rules will be covered, along with their encoding in KAT and proof that the encodings are theorems in KAT (from [7]).

3.2 Assignment rule

Definition 3.7 The assignment rule of Hoare logic is

$$\{b[x/e]\}x:=e\{b\} \tag{3.24}$$

As mentioned before, since KAT is purely propositional, there is no domain of computation and this rule is not considered in KAT.

3.3 Composition rule

The composition rule of Hoare logic is

Definition 3.8

$$\frac{\{b\}p\{c\}\qquad\{c\}q\{d\}}{\{b\}p\,;q\{d\}} \tag{3.25}$$

Lemma 3.9 The translation of 3.25 in KAT is also a theorem in KAT:

$$bp = bpc \wedge cq = cqd \to bpq = bpqd \tag{3.26}$$

Proof. Assuming the premises

$$bp = bpc \tag{3.27}$$
$$cq = cqd \tag{3.28}$$

we derive

$$\begin{aligned} bpq &= bpcq && \text{by } 3.27 \\ &= bpcqd && \text{by } 3.28 \\ &= bpqd && \text{by } 3.27 \end{aligned}$$

□

3.4 Conditional rule

The conditional rule of Hoare logic is

Definition 3.10

$$\frac{\{b \wedge c\}p\{d\} \qquad \{\neg b \wedge c\}q\{d\}}{\{c\}\textbf{if } b \textbf{ then } p \textbf{ else } q\{d\}} \tag{3.29}$$

Lemma 3.11 The translation of 3.29 in KAT is also a theorem in KAT:

$$bcp = bcpd \wedge \overline{b}cq = \overline{b}cqd \to c(bp + \overline{b}q) = c(bp + \overline{b}q)d \tag{3.30}$$

Proof. Assuming the premises

$$bcp = bcpd \tag{3.31}$$
$$\overline{b}cq = \overline{b}cqd \tag{3.32}$$

we derive

$$\begin{aligned} c(bp + \overline{b}q) &= cpb + c\overline{b}q && \text{distributivity} \\ &= bcp + \overline{b}cq && \text{commutivity of tests} \\ &= bcpd + \overline{b}cqd && 3.31 \text{ and } 3.32 \\ &= cbpd + c\overline{b}qd && \text{commutivity of tests} \\ &= c(bp + \overline{b}q)d && \text{distributivity} \end{aligned}$$

□

3.5 While rule

The while rule of Hoare logic is

Definition 3.12

$$\frac{\{b \wedge c\}p\{c\}}{\{c\}\textbf{while } b \textbf{ do } p\{\neg b \wedge c\}} \tag{3.33}$$

Lemma 3.13 The translation of 3.33 in KAT is also a theorem in KAT:

$$bcp = bcpc \rightarrow c(bp)^*\overline{b} = c(bp)^*\overline{b}\overline{b}c \tag{3.34}$$

Proof. Because all tests commute, and $=$ implies $\leq$, it suffices to prove

$$cbp \leq cbpc \rightarrow c(bp)^* \leq c(bp)^*c \tag{3.35}$$

Assuming the premise

$$cbp \leq cbpc \tag{3.36}$$

we get, using equation 2.4, that it suffices to show that

$$c + c(bp)^* \leq c(bp)^*c \tag{3.37}$$

We can now finish the proof:

$$\begin{align} c + c(bp)^* &\leq c(bp)^*c \tag{3.38}\\ &\leq c1c + c(bp)^*cbpc \tag{3.39}\\ &\leq c(1 + (bp)^*cbp)c \tag{3.40}\\ &\leq c(+(bp)^*bp)c \tag{3.41}\\ &\leq c(bp)^*c \tag{3.42}\end{align}$$

□

3.6 Weakening rule

The weakening rule of Hoare logic is

Definition 3.14

$$\frac{b' \rightarrow b \qquad \{b\}p\{c\} \qquad c \rightarrow c'}{\{b'\}p\{c'\}} \tag{3.43}$$

Lemma 3.15 The translation of 3.43 in KAT is also a theorem in KAT:

$$b' \leq b \wedge bp = bpc \wedge c \leq c' \rightarrow b'p = b'pc' \tag{3.44}$$

Proof. First, we use Theorem 3.1 to rewrite Equation 3.44 to:

$$b' \leq b \wedge bp\overline{c} = 0 \wedge c \leq c' \rightarrow b'p\overline{c'} = 0 \tag{3.45}$$

which follows from the monotonicity of multiplication. □

Chapter 4

A Folk Theorem

In [6], Kozen defines a folk theorem on while programs as follows:

Theorem 4.1 Every program in **while** with boolean variables, as defined in Chapter 3, can be simulated by a **while** program with at most one **while** loop.

Furthermore, he proves the following theorem:

Theorem 4.2 Every **while** program, suitably augmented with finitely many new subprograms of the form $s; bc + \overline{b}\overline{c}$, is equivalent to a **while** program in normal form, reasoning in Kleene algebra with tests under certain commutativity assumptions.

The remark about certain commutativity assumptions might seem vague, but this will be specified when relevant. He also gives code transformations for the while-constructions that produce equivalent programs in normal form. We will define a function N on while, which normalizes while programs. In the following, we will define the more trivial while statements, and in the remainder of this section, Kozens program transformations will be explained and proven correct in detail.

4.1 Normalizing While programs

Definition 4.3 Let N be a function on while-programs that takes a program and brings it to its normal form. We have:

$$N(x := a) = x := a; \textbf{while } \texttt{false} \textbf{ do } \texttt{skip} \tag{4.1}$$

$$N(\texttt{skip}) = \texttt{skip}; \textbf{while } \texttt{false} \textbf{ do } \texttt{skip} \tag{4.2}$$

$$N(S) = S \qquad \text{if } S \text{ in normal form} \tag{4.3}$$

The composition, conditional and while statements' transformations will be covered in the next sections.

4.2 Conditional program

For the conditional program

if b **then begin** p_1 ; **while** d_1 **do** q_1 **end**
    else **begin** p_2 ; **while** d_2 **do** q_2 **end**

he introduces a new test c that gets the value of b, assumes that c commutes with p_1, p_2, q_1 and q_2 (which we can assume since c is new), and transforms the new program

$c := b$
if b **then begin** p_1 ; **while** d_1 **do** q_1 **end**
    else **begin** p_2 ; **while** d_2 **do** q_2 **end**

to

$c := b$
if c **then** p_1 **else** p_2 ;
while $cd_1 + \bar{c}d_2$ **do**
  if c **then** q_1 **else** q_2

It's important to note that if both programs in the conditional (p_1 ; **while** d_1 **do** q_1 and p_2 ; **while** d_2 **do** q_2) are in normal form, then the resulting program will also be in normal form.

We can now define $N($**if** b **then** S_1 **else** $S_2)$ as the piece of code above, assuming S_1 and S_2 are in normal form.

4.3 Program with nested while loops

For a program containing nested while loops, he shows that the program

while b **do begin**
    p ;
    while c **do** q
  end

is, without the need of additional commutivity assumptions, equivalent to

if b **then begin**
    p ;
    while $b + c$ **do**
      if c **then** q **else** p
  end

which now contains only one while loop inside a conditional. This program is not in the right format for the transformation from Section 4.3, so we add a dummy else clause. We also need a commutivity assumption for b and therefore introduce a new test d, resulting in:

$d := c$
if b **then begin**
    p ;
    while $b + c$ **do**
      if c **then** q **else** p
  end
else begin 1 ; **while** 0 **do** 1 **end**

This program can be transformed into:

$d := c$
if d **then** p **else** 1 ;

```
while d(b + c) + d̄1 do
      if d then begin
                  while b + c do
                    if c then q else p
                  end
          else while 0 do 1
```

This means that the latter is also equivalent with the first program from this section, preceeded by $d := c$.

Again, it is important to notice that if the inner while loop is in normal form, then the resulting program will be as well.

We can now define N(**while** b **do** (p ; **while** c **do** q)) as the piece of code above.

4.4 Getting rid of postcomputations

For a program that contains a postcomputation after a while loop

```
while b do p ; q
```

he shows that, assuming b and p commute, it is equivalent to

```
if b̄ then q
    else while b do begin
                    p ;
                    if b̄ then q
                  end
```

We may assume that b and q commute; if they do not commute, we can introduce a new test c and atomic program s that sets the value of c to b, and insert s before the loop and after the body of the loop. It is proven in [6] that the program is now equivalent to a program where c is tested in the while loop.

We can once more transform the program above using the transformation from Section 4.2 (note that $\bar{b}$ commutes with c now) into:

```
b := c̄
if c then q else 1 ;
while c0 + c̄b do
  if c then 1 else p ; if b̄ then q
```

which reduces by Boolean logic and removal of the useless else-clause to:

```
b := c̄
if c then q
while bc̄ do
  if c̄ then
    p ;
    if b̄ then q
```

Yet again, it's important to note that if p and q do not contain a while loop, that the resulting program is in normal form and is the definition of N(**while** b **do** p ; q).

4.5 Composition of programs

Finally, he shows that the composition of two programs in normal form

```
p1 ;
while b1 do q1 ;
p2 ;
while b2 do q2 ;
```

can be transformed into one program in normal form. In order to do that, he states that p_2 can be sucked into the first while loop with the transformation of Section 4.4:

p_1 ;
$c := \overline{b_1}$;
if c **then** q_1
while $b_1\overline{c}$ **do**
  if $\overline{c}$ **then**
    p_2 ;
    if $\overline{b_1}$ **then** q_1
while b_2 **do** q_2 ;

For readability, the body of the first while loop is abbreviated to r and the precomputation, consisting of the first three lines, to p_0:

p_0 ;
while $b_1\overline{c}$ **do** r
while b_2 **do** q_2 ;

Using Kozen's transformation, we get

p_0 ;
if $\overline{b_1}$ **then while** b_2 **do** q_2
  else while b_1 **do begin**
      r ;
      if $\overline{b_1}$ **then while** b_2 **do** q_2
    end

If we substitute p_0 and r out, we get:

p_1 ;
$c := \overline{b_1}$;
if c **then** q_1
if $\overline{b_1}$ **then while** b_2 **do** q_2
  else while b_1 **do begin**
      if $\overline{c}$ **then**
        p_2 ;
        if $\overline{b_1}$ **then** q_1
      if $\overline{b_1}$ **then while** b_2 **do** q_2
    end

Using the transformations described in the earlier sections, a program in normal form will result from repeated transformations.

Chapter 5

Conclusion

In the previous sections, a number of transformations have appeared; Figure 5.1 intends to vizualize the relation between the different statements. The "basic" statement S can be seen in the top left corner. The horizontal arrow at the top is the [ ]-function from Definition 3 that translates a while-program to KAT. The vertical arrow on the left represents the normalization function N from Definition 4.1. The horizontal arrow at the bottom is again the [ ]-function, now applied on a normalized program. The big equals sign on the right represents the equality between the programs KAT-term and the normalized programs KAT-term, which is indeed proven.

In his work, Kozen ignores the assignment rule, since his reasoning system (Kleene Algebra with Tests) is purely propositional. While that is correct, we do feel that it is missing in the big picture, since it makes it impossible to prove the equality of the code and the normalized code of any significant while-program. To overcome this, we introduced the KAT-constants from Definition 3, and showed that it is now possible to reason and make (or translate) deductions, be it that there is now a number of constants in the assumptions.

Future research could be done in order to come up with a more elegant way to deal with the assignment problem.

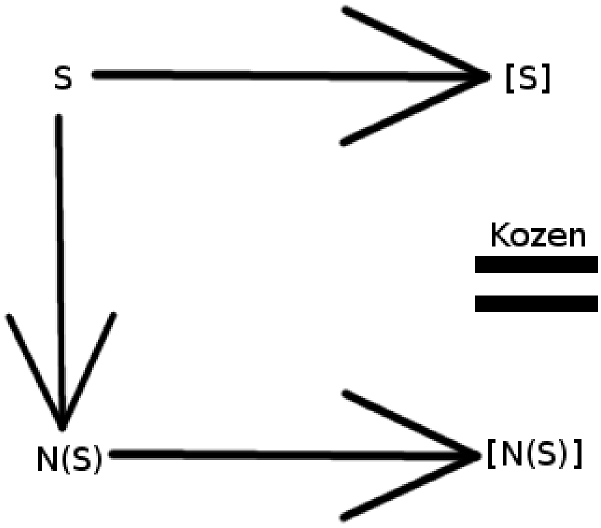

Figure 5.1: Schema of the transformations

Bibliography

[1] Conway, J. H. *Regular Algebra and Finite Machines.* MATHEMATICS SERIES Series. John Wiley & Sons, Incorporated, 1973.

[2] Davey, B. A., and Priestly, H. A. *Introduction to Lattices and Order.* Cambridge University Press, 1990.

[3] Desharnais, J., Möller, B., and Struth, G. Kleene algebra with domain. *ACM Trans. Comput. Log. 7*, 4 (2006), 798–833.

[4] Harel, D., Kozen, D., and Tiuryn, J. *Dynamic logic.* FOUNDATIONS OF COMPUTING SERIES. Mit Press, 2000.

[5] Hoare, C. A. R. An axiomatic basis for computer programming. *Commun. ACM 12*, 10 (Oct. 1969), 576–580.

[6] Kozen, D. Kleene algebra with tests. *Transactions on Programming Languages and Systems 19*, 3 (May 1997), 427–443.

[7] Kozen, D. On Hoare logic and Kleene algebra with tests. *Trans. Computational Logic 1*, 1 (July 2000), 60–76.

[8] Nielson, H. R., and Nielson, F. *Semantics with applications: a formal introduction.* John Wiley & Sons, Inc., New York, NY, USA, 1992.

[9] Pratt, V. Dynamic algebras as a well-behaved fragment of relation algebras. In *Algebraic Logic and Universal Algebra in Computer Science*, C. Bergman, R. Maddux, and D. Pigozzi, Eds., vol. 425 of *Lecture Notes in Computer Science.* Springer New York, 1990, pp. 77–110.