Universiteit Leiden
Opleiding Informatica

Describing heaps using Kleene algebra with tests

Name:	Ulbe van der Werf
Studentnr:	1166328
Date:	June 29, 2016
1st supervisor:	Marcello Bonsangue
2nd supervisor:	Frank de Boer

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Describing heaps using Kleene Algebra with tests

Ulbe van der Werf

Abstract

We explore a method for describing heaps using Kleene algebra with tests by reinterpreting heaps as KAT-automata. Using this reinterpretation we test the inclusion of a heap in a KAT-expression, explore the minimal conditions for a KAT-expression to accept any heaps, and weaken the used method of testing inclusion to simplify the expressions needed to describe a heap. We provide a tool to generate automata from KAT-expressions, and to test whether a given heap is accepted by a given expression.

Contents

Chapter 1

Introduction

Heaps are powerful data structures where objects may hold the location of other objects in their properties. A linked list can be constructed by having each object point to the next object in line or a tree structure could be constructed by having each object point to a number of child objects. Large sections of such structures can be moved by changing the target of one pointer. The power of heap structures also poses a risk when it becomes unclear which object is being altered. After several steps across pointers it becomes possible to inadvertently create loops or detach a subheap from the main structure. To mitigate these problems we propose a way to specify heaps using Kleene algebra with tests. This method could be a first step to creating correctness proofs for programs on heaps.

Kleene algebra with tests

Kleene algebra with tests (KAT) is an extension to Kleene algebra. Kleene algebra is a method for defining languages using operators as choice, concatenation and repetition. An expression of Kleene algebra is also called a regular expression. KAT expands on this system by adding boolean tests. In a KAT-expression, each transition is guarded by a boolean test which must be satisfied by the input to continue along that path. KAT is more commonly used to model Hoare logic in program correctness proofs. An example of this is [Kozoo].

Testing KAT acceptance

We use the method to reinterpret heaps as KAT-automata proposed by [Joc13] and implement their method for deciding inclusion of such a heap in a given KAT-expression. This method uses bisimulation up to congruence. A method which creates a relation between sets of nodes that produce the same output in different

automata. We will expand this interpretation of acceptance to create more workable KAT-expressions to denote heaps.

1.0.1 Overview

Chapter 2 provides definitions for Kleene algebra with tests, heaps and bisimulations of KAT-expressions. Chapter 3 discusses several publications which inspired this work. Chapter 4 proposes a modification to [Joc13]; introduces a tool we wrote to automate automaton generation and testing of heap inclusion; explores the minimum requirements for a KAT-expression to denote a heap; and explores two looser ways to check whether a heap satisfies an automaton. Chapter 5 concludes.

Chapter 2

Definitions

2.1 Kleene algebra with tests [ABM12] [Joc13] [Koz00]

A Kleene algebra with tests is a Kleene algebra $K = (\Sigma, \cdot, +, 0, 1)$ such that there is a subset B of K for which there exists a boolean algebra $B = (\Upsilon, \cdot, +, \neg, 0, 1)$. Here $\neg$ represents negation. We will interpret elements of Υ as statements on the identity of the current object, and elements of Σ as navigation expressions that move us from an object to another object.

Given a finite set $\Sigma = \{p_1, p_2, ..., p_n\}$ of primitive actions and a finite set $\Upsilon = \{t_1, t_2, ..., t_m\}$ of primitive tests, the set $BExp$ of boolean expressions generated by Υ is

$$b ::= 0 \,|\, 1 \,|\, t \in \Upsilon \,|\, \bar{b} \,|\, b + b \,|\, b \cdot b \,.$$

The set Exp of regular expressions extended with boolean tests is given by

$$e ::= p \in \Sigma \,|\, b \in BExp \,|\, e + e \,|\, e \cdot e \,|\, e^* \,.$$

In addition, we define the set of atoms, At, of all possible valuations of the tests in Y. Because each atom must assign true or false to each test, its size is 2^Y. Because each expression in $BExp$ must either accept or not accept each atom, the number of logically distinct boolean expressions is $2^{|At|} = 2^{2^Y}$. We say that an atom $a \in At$ proves a boolean expression $b \in BExp$ if, and only if, $a \vDash b$. We write $a \vdash b$.

We define the semantics of KAT-expressions in terms of guarded strings. These are strings of the form $\alpha_1 p_1 \alpha_2 p_2 ... \alpha_n p_n \alpha_{n+1}$. More formally the set of guarded strings GS is given by $GS = (At \times \Sigma)^* \times At$. Using the interpretation given earlier these strings can be considered to describe walks through a heap. We define

the guarded language $G(e) \subseteq GS$ generated by e inductively as

$$G(p) = \{\alpha_1 p \alpha_2 | \alpha_1, \alpha_2 \in At\}$$
$$G(b) = \{\alpha | \alpha \in At \wedge \alpha \vdash b\}$$
$$G(e_1 + e_2) = G(e_1) \cup G(e_2)$$
$$G(e_1 \cdot e_2) = G(e_1) \star G(e_2)$$
$$G(e^*) = \cup_{n \in \mathbb{N}} G(e)^n$$

Where we define $G_1 \star G_2$ as $\{v\alpha w | v\alpha \in G_1 \wedge \alpha w \in G_2\}$, $G^0 = At$, and $G^n = G^{n-1} \star G$ for $n \in \mathbb{N}_{>0}$. A guarded language $G \subseteq GS$ is said to be regular if and only if there exists a KAT-expression such that $G(e) = G$.

2.1.1 Constructing non-deterministic finite automata for KAT

Automata on guarded strings

We construct non-deterministic finite automata for Kleene Algebra with Tests (KAT) according to the method described by [Joc13]. We have been helped in understanding this method by similar constructions in [Sil12] and [ABM12]. This method defines a partial derivative function for KAT-expressions and constructs automata for them in which each partial derivative of the original expression is a node. We will propose a modification to this method in 4.1.

Given an alphabet of actions Σ, and a set of atomic tests τ we define a non-deterministic automaton over guarded strings as a tuple $M =< Q, \Delta, \theta, I >$ consisting of a set Q of states, a transition function Δ, an immediate output function θ, and a set of initial states I with $I \subseteq Q$.

The function Δ is of the form:

$$\Delta : Q \to \mathcal{P}(Q)^{BExp \times \Sigma} \tag{2.1}$$

where $BExp$ denotes the set of boolean expressions over τ. Δ defines the transitions that are available from a node while reading an element of $BExp$ and an element of Σ. Note that each transition goes to a set of states. Because of this, it is possible for an NFA to be in multiple states simultaneously.

The function θ, which assigns a boolean expression to elements of Q takes the form:

$$\theta : Q \to BExp \tag{2.2}$$

A state q accepts while reading input a if, and only if, $a \vdash \theta(q)$. Note that the set $BExp$ is technically infinite but we will only consider its normalized, logically distinct members here.

We define the recursive function $G(M)(q)$ for $a, w \in At$ and $p \in \Sigma$, which determines the input accepted by the automaton M from state $q \in Q$ as:

$$a \in G(M)(q) \Leftrightarrow a \vdash \theta(q) \tag{2.3}$$

$$apw \in G(M)(q) \Leftrightarrow \exists b : (a \vdash b \wedge \exists q \xrightarrow{b,p} q' \wedge w \in G(M)(q')) \tag{2.4}$$

We may then define the set $G(M)$ of guarded strings accepted by M as:

$$G(M) = \bigcup_{q \in I} G(M)(q) \tag{2.5}$$

Transforming KAT-expressions to NFA

To transform KAT-expressions to NFA we first define two functions, $out(e)$ and $first(e)$, that define the immediate output and the first continuation of an expression e respectively. We will use $out(e)$ to determine whether the current node is accepting and, if it is accepting, with what output we can terminate. We will use $first(e)$ to generate the transitions in our automaton.

$$out(b) = b \tag{2.6}$$

$$out(p) = 0 \tag{2.7}$$

$$out(e_1 + e_2) = out(e_1) + out(e_2) \tag{2.8}$$

$$out(e_1 e_2) = out(e_1) \cdot out(e_2) \tag{2.9}$$

$$out(e^*) = 1 \tag{2.10}$$

Using out we define that for each $e \in Exp, B \in BExp$ and $p \in \Sigma$, the partial derivatives $\delta_{bp}(e)$ of e are:

$$\delta_{bp}(b) = \emptyset \tag{2.11}$$

$$\delta_{bp}(q) = \begin{cases} 1 & \text{if } p = q \\ \emptyset & \text{otherwise} \end{cases} \tag{2.12}$$

$$\delta_{bp}(e_1 + e_2) = \delta_{bp}(e_1) \cup \delta_{bp}(e_2) \tag{2.13}$$

$$\delta_{bp}(e_1 \cdot e_2) = \begin{cases} \delta_{bp}(e_1) \cdot e_2 \cup \delta_{bp}(e_2) & \text{if } out(e_1) \neq 0 \\ \delta_{bp}(e_1) \cdot e_2 & \text{otherwise} \end{cases} \tag{2.14}$$

$$\delta_{bp}(e^*) = \delta_{bp}(e) \cdot e^* \tag{2.15}$$

$$\text{Where } X \cdot e' = \{ee' | e \in X\} \tag{2.16}$$

To prevent the generation of unneeded states, we will only generate partial derivatives with regard to the first possible continuation of a guarded string from an expression e. To determine what these are we define the function $first(e)$ as follows:

$$first(b) = \emptyset \tag{2.17}$$

$$first(1, p) = (1, p) \tag{2.18}$$

$$first(e_1 + e_2) = first(e_1) + first(e_2) \tag{2.19}$$

$$first(e_1 e_2) = first(e_1) \cup out(e_1) \triangleright first(e_2) \tag{2.20}$$

$$first(e^*) = first(e) \tag{2.21}$$

Using these functions we can define the automaton $M(e) =< Q, \Delta, \theta, I >$ as follows:

$$Q = \bigcup \{\delta_w(e) | w \in (BExp * \Sigma)^*\} \tag{2.22}$$

$$\Delta(q)(b, p) = \begin{cases} \delta_{bp}(q) & \text{if } (b, p) \in first(q) \\ \{\} & \text{otherwise} \end{cases} \tag{2.23}$$

$$\theta(q) = out(q) \tag{2.24}$$

$$I = e \tag{2.25}$$

An example

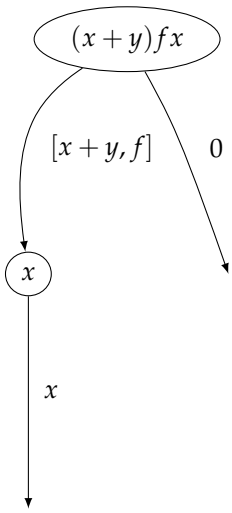
To demonstrate we will generate an automaton M_e for the KAT-expression $e = (x + y)fx$ according to 2.1.1. We first generate e's partial derivatives.

$$\begin{aligned}
first((x+y)fx) &= first(x+y) \cup out(x+y) \triangleright first(fx) \\
&= \emptyset \cup (x+y) \triangleright (first(f) \cup out(f) \triangleright first(x)) \\
&= (x+y) \triangleright (\{(1,f)\} \cup 0 \triangleright \emptyset) \\
&= \{(x+y,f)\} \\
out((x+y)fx) &= out(x+y) \cdot out(fx) \\
&= (x+y) \cdot out(f) \cdot out(x) \\
&= (x+y) \cdot 0 \cdot x \\
&= 0 \\
\delta_{(x+y)f}((x+y)fx) &= \delta_{(x+y)f}(x+y) \cdot fx \cup \delta_{(x+y)f}(fx) \\
&= (\delta_{(x+y)f}(x+y) \cdot fx \cup \delta_{(x+y)f}(f) \cdot x \\
&= \emptyset \cdot fx \cup 1 \cdot x \\
&= x \\
first(x) &= \emptyset \\
out(x) &= x
\end{aligned}$$

Note that because $first(x)$ is empty, we do not need to calculate the derivative of x as it will not be a state in our automaton. We can now construct M_e:

$$\begin{aligned}
Q_e &= \{e, x\} \\
\Delta_e &= \begin{cases} (e, < x, f >) \to x \\ (e, < y, f >) \to x \end{cases} \\
\theta_e &= \begin{cases} e \to 0 \\ x \to x \end{cases} \\
I_e &= \{e\}
\end{aligned}$$

Represented graphically:



2.2 Heaps [Joc13]

2.2.1 Definition

A heap is a structure consisting of a set of objects. Each object has a number of fields which refer to objects in the heap. Note that this does not need to be a different object. We represent the object being pointed to by an object x over a field *next* by $x.next$. Using these fields it is possible to create complex structures inside a heap. A common example is a linked list, in which each object points to the next object in line. Using such a structure a program only needs to keep track of the first object in line. Accessing all the following objects is done by repeatedly following the *next* pointer (or whatever the pointer being used is called). Inserting an object y into such a list, after object x, is done as follows: $y.next := x.next; x.next := y;$. This provides an advantage over inserting items into an array where all items after the inserted item need to be moved to make room for the new item.

We define heaps over a finite set of variables Var, and a finite set of fields Fld. A heap H is given by a tuple $H =< X, h, s >$. In this tuple, $X = \{x_1, x_2, ...\}$ is the set of all objects in our heap. It contains one distinguished element nil. The function h is used to model the fields. The function s maps our program variables to objects in the heap. Formally: h is an assignment $h : Fld \to (X \to X)$, and s is an assignment $s : Var \to X$. We also require that $h(f)(s(nil)) = s(nil)$, that is, all fields of the nil object point to itself.

Note that we have implicitly defined two properties of heaps because proper functions have exactly one output for each input. Firstly, because $s(v)$ yields exactly one object $x \in X$, each label refers to exactly one object. Secondly, because the functions produced by h have this property, each field of each object points to exactly one object. For convenience sake, we will sometimes omit defining some labels or fields that point to nil. Note that these properties do not imply that each object has exactly one label, or that each object is

pointed to by exactly one field. Each object may have any number of labels and may be pointed to by any number of fields.

2.2.2 Interpreting heaps as NFA

Because heaps and non-deterministic automata for KAT-expressions are both essentially automata with states and labeled transitions it is possible to interpret one as the other. Specifically, we can interpret heaps as KAT-automata. Intuitively, we will consider the states of our heap as the states of an automaton. The transitions in this automaton will be the fields of our heap, guarded by the label of the node we are departing from.

We can interpret a heap $H =< X, s, h >$ as a guarded string automaton $M =< Q, \delta, \theta, I >$ as follows:

1. The set of atomic tests At is the set of variables Var,
2. the set of actions Σ is the set of fields Fld,
3. the set of states Q is the set of objects X,
4. the set of initial states I is the set $I = \{x \in X | \exists_{v \in Var} : s(v) = x\}$.
5. The functions θ and δ are given by:

$$v \in \theta(x) \Leftrightarrow s(v) = x \tag{2.26}$$

$$x' \in \delta(x)(b, f) \Leftrightarrow \forall_{v \in \theta(x)} : v \vdash b \wedge h(f)(x) = x' \tag{2.27}$$

2.3 Bisimulations for Kleene algebra with tests

A bisimulation between sets of states in one automaton and sets of states in another automaton. For each pair, the set of states in the first automaton will produce the same output as the set of states in the second automaton. Generating a bisimulation is done by choosing a starting node in the first automaton and attempting to find a set of states that produces the same output in the second automaton. Usually, two sets of states will only be bisimilar if two other sets of states are bisimilar. This recursion will traverse the possible paths from the starting node to some set of states that has already been proven to be bisimilar. Our implementation is similar to [Joc13] and [Pou14].

We keep track of two lists, *Todo* and *Done*. To decide whether the language $G(H)(q)$ of a heap H from a state q is included in the language $G(e)$ of a KAT-expression e we insert $[q, e]$ into *Todo*. We then iteratively take a pair $[Q, E]$ from *Todo* and perform the following steps on it until *Todo* is empty or a test fails.

1. Test that $out(Q) \vdash out(E)$.
2. For each transition f from Q add $[Q.f, \delta_f(E)]$ to *Todo* if it is not already in *Done*
3. Add $[Q, E]$ to *Done*

If a test fails, the heap is not included in the automaton. If the process terminates because *Todo* is empty, there is a bisimulation between the heap and automaton. The sets of bisimilar states is contained in *Done*.

Chapter 3

Related Work

3.1 Partial derivatives for KAT expressions [Joc13]

This paper sets out to construct an efficient method for proving equivalence between KAT expressions. They use partial derivatives to construct non-deterministic finite automata for KAT expressions. Their equivalence proofs depend on bisimulation up to congruence. We borrow many of the methods described in this paper and suggest adaptations to improve its suitability for our purpose.

3.2 Automated Verification of Recursive Programs with Pointers [FdBR12]

This paper develops a dialect of propositional dynamic logic for describing heap structures. Using this logic they formulate a systematic way of generating the strongest postcondition belonging to a precondition and an assignment. This method is used to prove the correctness of annotated recursive programs with pointers. A possible continuation of our work would be to adapt the methods described in this paper to work with KAT-expression rather than propositional dynamic logic.

Chapter 4

Contributions

4.1 Parsing boolean expressions in δ

4.1.1 Incorrect transitions in δ

Before we proceed to automating the generation of automata for KAT-expressions we must first make a modification to the definition provided in 2.1.1. The current definition of δ does not take into account the boolean expression with which we are taking a derivative. This causes our automaton to have undesirable transitions. To demonstrate we will generate the derivatives of $e = xfy + yfz$:

First we generate $first(e)$:

$$
\begin{aligned}
first(xfy + yfz) &= first(xfy) \cup first(yfz) \\
first(xfy) &= first(x) \cup out(x) \triangleright first(fy) \\
&= \emptyset \cup x \triangleright (first(f) \cup out(f) \triangleright first(y)) \\
&= x \triangleright (\{[1, f]\} \cup 0 \triangleright \emptyset) \\
&= x \triangleright \{[1, f]\} \\
&= \{[x, f]\} \\
first(yfz) &= \ldots = \{[y, f]\} \\
first(xfy + yfz) &= \{[x, f], [y, f]\}
\end{aligned}
$$

Before we generate $\delta(xfy + yfz)$ We reproduce the definition of δ here for convenience:

$$\delta_{bp}(b) = \emptyset \tag{4.1}$$

$$\delta_{bp}(q) = \begin{cases} 1 & \text{if } p = q \\ \emptyset & \text{otherwise} \end{cases} \tag{4.2}$$

$$\delta_{bp}(e_1 + e_2) = \delta_{bp}(e_1) \cup \delta_{bp}(e_2) \tag{4.3}$$

$$\delta_{bp}(e_1 \cdot e_2) = \begin{cases} \delta_{bp}(e_1) \cdot e_2 \cup \delta_{bp}(e_2) & \text{if } out(e_1) \neq 0 \\ \delta_{bp}(e_1) \cdot e_2 & \text{otherwise} \end{cases} \tag{4.4}$$

$$\delta_{bp}(e^*) = \delta_{bp}(e) \cdot e^* \tag{4.5}$$

$$\text{Where } X \cdot e' = \{ee' | e \in X\} \tag{4.6}$$

We will now generate $\delta_{bf}(xfy)$ with b some boolean expression. Problematically, we do not need to specify whether $b \vdash x$ or $b \nvdash x$.

$$\delta_{bf}(xfy) = \delta_{bf}(x) \cdot fy \cup \delta_{bf}(fy) \qquad \text{because } out(b) \neq 0 \tag{4.7}$$

$$= \emptyset \cdot fy \cup \delta_{bf}(f) \cdot y \qquad \text{because } out(f) = 0 \tag{4.8}$$

$$= \{1\} \cdot y \tag{4.9}$$

$$= \{y\} \tag{4.10}$$

Because we did not specify the actual value of b we similarly have that:

$$\delta_{xf}(xfy) = \{y\} \tag{4.11}$$

$$\delta_{xf}(yfz) = \{z\} \tag{4.12}$$

$$\delta_{yf}(xfy) = \{y\} \tag{4.13}$$

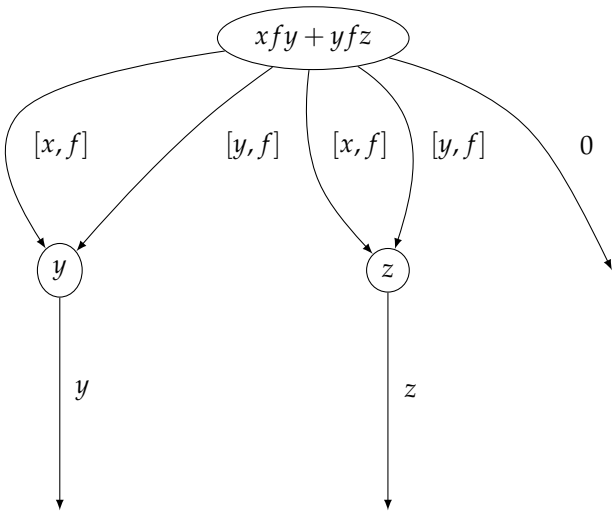
$$\delta_{yf}(yfz) = \{z\}. \tag{4.14}$$

And therefore:

$$\delta_{xf}(xfy + yfz) = \{y, z\} \tag{4.15}$$

$$\delta_{yf}(xfy + yfz) = \{y, z\}. \tag{4.16}$$

If we construct an automaton using these results we get the following:



This automaton correctly accepts xfy and yfz, but it also incorrectly accepts xfz and yfy. This is caused by the missing check in δ.

4.1.2 Testing for satisfaction in δ

The problem described in the previous section occurs because 4.4 only tests whether the expression has an output. Not whether that output satisfies the boolean expression with regard to which we are taking the derivative. However, it is also incorrect to demand that $out(e_1) \vdash b$ as can be seen in the following example:

$$\begin{aligned}\delta_{xyf}(x(yfz)) &= \delta_{xyf}(x) \cdot yfz \qquad \text{because } out(x) \nvdash xy \\ &= \emptyset \cdot yfz \\ &= \emptyset\end{aligned}$$

We propose to modify two lines of δ. Firstly the recursive case for $\delta_{bp}(e_1 \cdot e_2)$ should keep track of the tests that have already been satisfied since the last navigation expression by replacing them with 1. Secondly, the derivative of f should only be 1 if the boolean guard is a tautology. Formulaically:

$$\delta_{bp}(q) = \begin{cases} 1 & \text{if } p = q \wedge b \equiv 1 \\ \emptyset & \text{otherwise} \end{cases} \tag{4.17}$$

$$\delta_{bp}(e_1 \cdot e_2) = \begin{cases} \delta_{b'p}(e_1) \cdot e_2 \cup \delta_{b'p}(e_2) & \text{if } out(e_1) \neq 0 \\ \delta_{b'p}(e_1) \cdot e_2 & \text{otherwise} \end{cases} \tag{4.18}$$

$$\text{Where } b' = b[e_1/1] \tag{4.19}$$

Using this modified definition we can calculate the following values for $\delta_{xf}(xfy)$ and $\delta_{xf}(yfz)$:

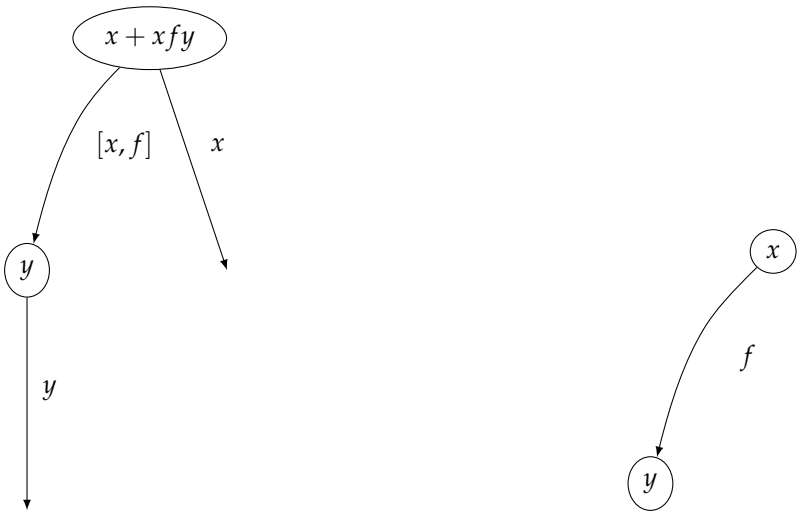
$$\begin{aligned}
\delta_{xf}(xfy) &= \delta_{x[x/1]f}(x) \cdot fy \cup \delta_{x[x/1]f}(fy) \\
&= \emptyset \cdot fy \cup \delta_{1f}(f) \cdot y \\
&= 1 \cdot y \\
&= y \\
\delta_{yf}(xfy) &= \delta_{y[x/1]f}(x) \cdot fy \cup \delta_{y[x/1]f}(fy) \\
&= \emptyset \cdot fy \cup \delta_{yf}(f) \cdot y \\
&= \emptyset \cdot y \\
&= \emptyset.
\end{aligned}$$

4.2 KAT-automata that do not specify any heaps

Since it is possible to interpret heaps as KAT-automata we can evaluate whether the language of a given heap and a given label is accepted by a KAT-automaton. Although a KAT-automaton can be generated for every heap, it is not possible to find an accepted heap for every KAT-automaton. We will investigate some of the properties of a KAT-automaton which make it impossible to find a heap which produces a sublanguage of it.

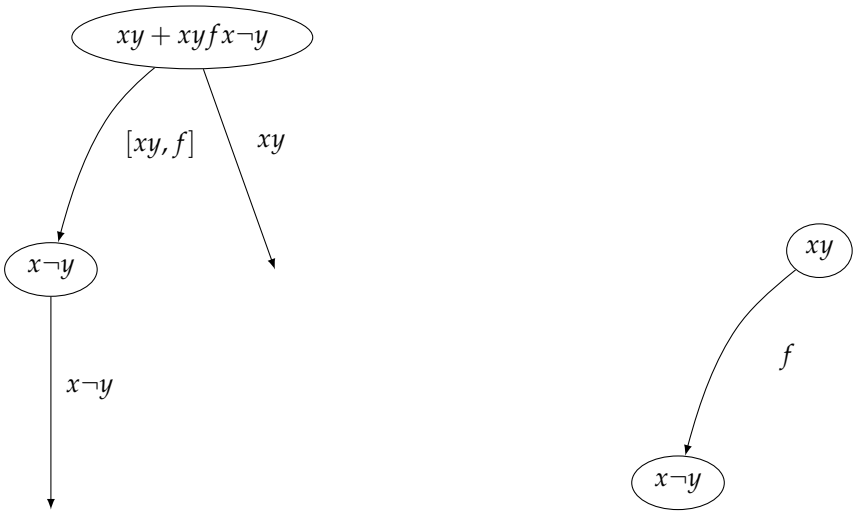
Automata that do not accept their own substrings

In a heap, every intermediate object is a valid point to terminate a walk. It is impossible for a transition to only be valid if it is followed by another one. Because of this a heap will always accept all substrings of strings it accepts. KAT-automata do not have this restriction. Automata which do not satisfy this restriction can never be matched to a heap. For example, the expression xfy accepts the string xfy, but not its substring x. A heap in which xfy is a valid walk, would also accept x.



Multiply occurring labels

In KAT-automata, each boolean expression exists in isolation. If a boolean expression does not contain a contradiction it can validate guarded expressions. In heaps the boolean expressions are interpreted as labels that indicate a specific object globally. Each labeling in a single object applies restrictions to the labels that can occur in all other objects. For example, a KAT-automaton can accept a string that contains the valuations xy and $x\overline{y}$. This simply indicates that both x and y are true at one point in the string and only x is true at another point. In a heap the first substring indicates that the current object is labeled both x and y. The second substring indicates the current object is labeled x, but not labeled y. Since labels are global, this entails that this object is labeled y and $\overline{y}$ which is a contradiction.



Missing transitions

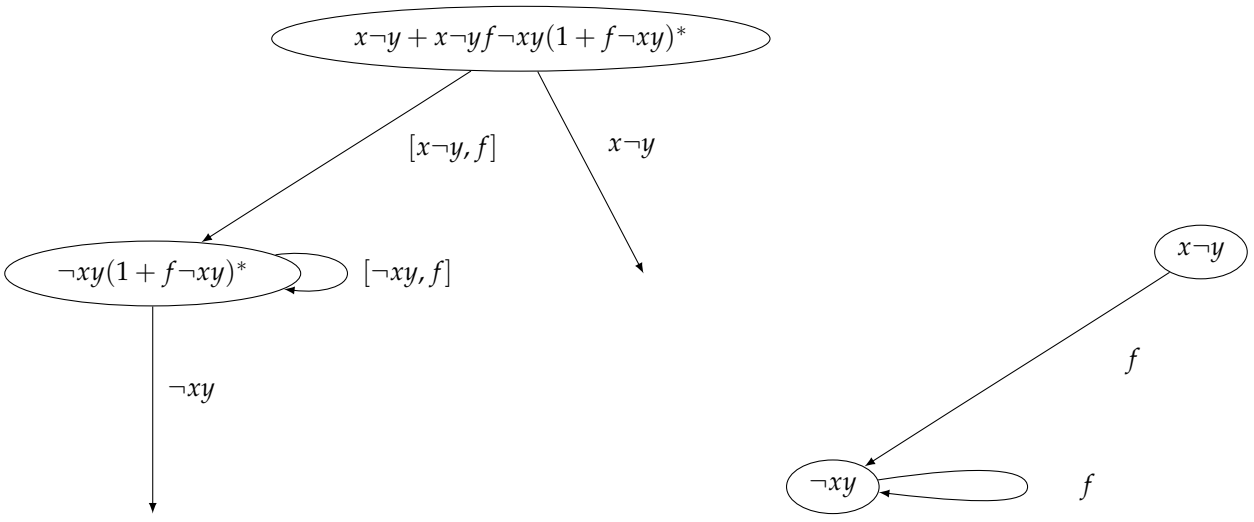
In a heap, each object has a, possibly null, object indicated in each of its fields. This means that a heap always has a possible continuation over each field from each object. KAT-automata which fail to specify a target for each combination of transition and node cannot accept the same language as a heap. All the heaps shown in this section suffer from this problem.

Finite KAT-automata

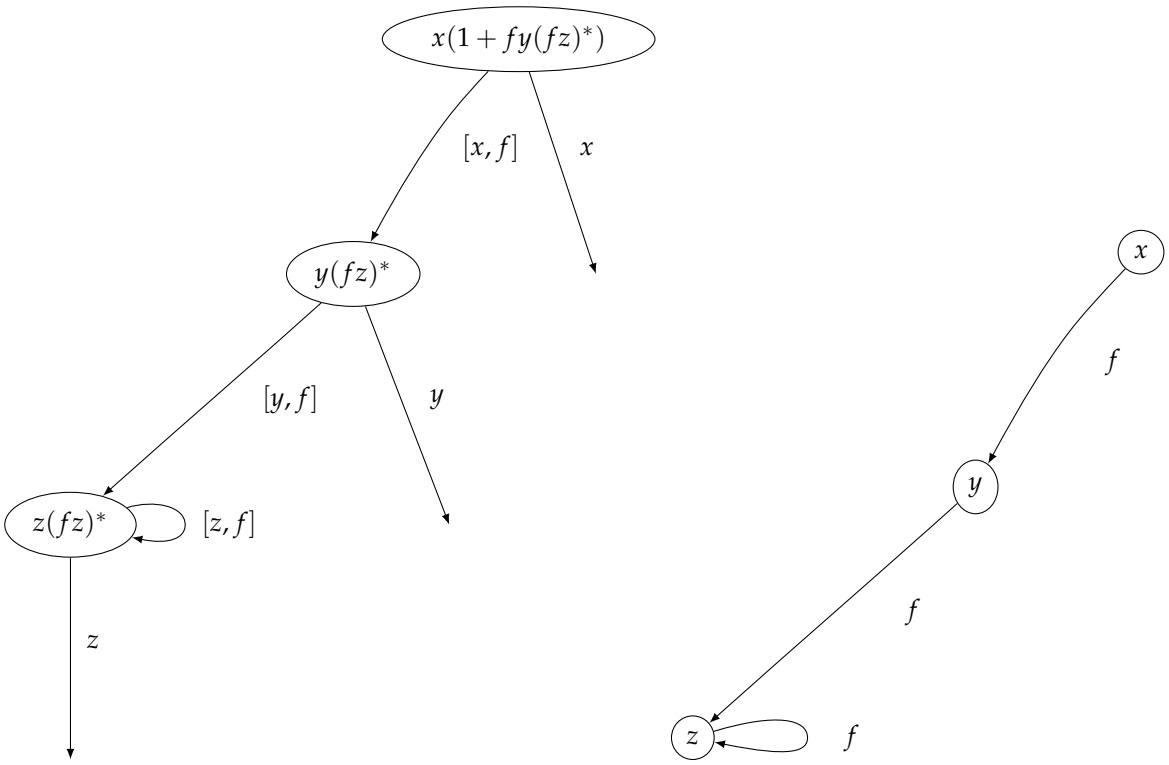
As stated, all fields of all objects in a heap must point somewhere. Because heaps are finite structures, the fields cannot point forward in an infinite regression. This means that every heap must contain at least one loop. For some heaps this will only be the loop the null object has to itself over all fields, but for others more complex loops may exist. Regardless, because of this loop the language accepted by a heap is always infinite. It does not matter that this infinite language may consist of strings of the form $\emptyset f \emptyset f \emptyset \ldots$ Any KAT-automaton that does not accept an infinite language cannot be a heap. All the heaps shown in this section suffer from this problem as well.

KAT-automata that can be satisfied

We finish this section with some examples of KAT-automata which do have heaps that satisfy them.



We see that, starting from $x\overline{y}$ both the automaton and the heap presented here can terminate, or they can continue while reading f. If they do this they go to a state where they can terminate with $\overline{x}y$, or continue while reading another f and returning to the same node.



These automata, starting from x, can terminate or read f to transition to y. From y they can terminate or transition over f to z. Note that the automaton given here could accept strings which the heap could not accept, such as $x\overline{y}fxy$. This is not a problem since all strings accepted in the heap from x are accepted by the automaton.

4.3 Automating automaton generation

To simplify the process of generating heaps we have implemented the automaton generation technique outlined in 2.1.1 in a Prolog tool. The next section makes extensive use of this tool. Because some symbols are reserved by Prolog, and others simply cannot be input easily we convert our KAT-expressions in the following way:

- xy or $x \cdot y$ becomes `x*y`
- e^* becomes `e**`.
- $\neg x$ becomes `~x`.
- It is sometimes necessary to add spaces to delimit terms. e.g. `x+ y**` rather than `x+y**`.

4.4 Testing whether a heap satisfies an expression

We have seen several examples of expressions that cannot be satisfied by any heap. To further explore the relationship between heaps and expressions we will investigate a number of expressions and heaps that may satisfy them. We will use the program described in the previous section to decide whether a heap satisfies an automaton. Before we investigate specific cases we will describe the syntax of a program call.

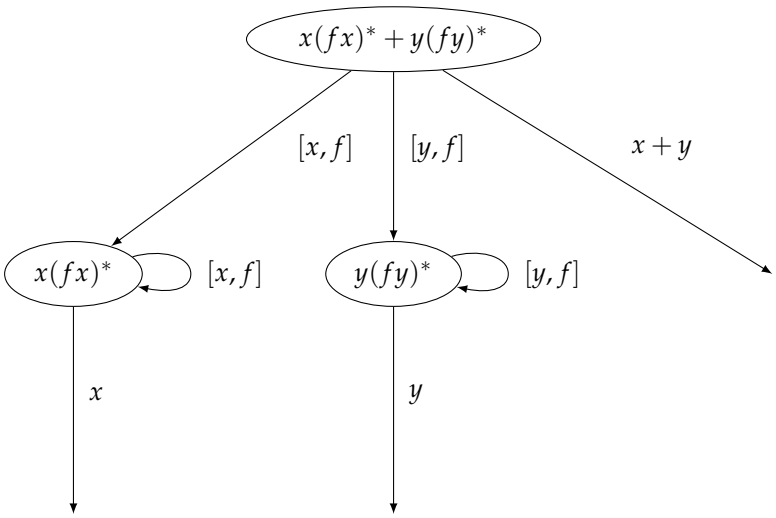
Syntax of `sat()`

A call to `sat()` takes the following form: `sat(States, Transitions, Labels, Start, E)`. Where `States` is the set of states X in our heap, `Transitions` is the transition function h, `Labels` is the labeling function θ, `Start` is the starting object in our heap, and `E` is the expression for which we are testing inclusion. If the call is successful it will print the bisimulation that was constructed to satisfy our expression. Recall that we use a different syntax (`*` for product, `**` for Kleene star, and `~b` for negation) to accommodate the syntax of the Prolog programming language.

In some examples we will omit some transitions from a heap so we can use a simpler expression and obtain a simpler result. It is important to note that these heaps are technically invalid because a heap must always have exactly one target for each transition of each object. Because `sat()` interprets the given heap as a nondeterministic automaton it does not check these restrictions.

Example of a call to `sat()`

We will show a call to `sat()` to decide whether node x of the following heap satisfies the following automaton:



The call to test this inclusion is:

```
sat([q1,q2], [[q1,f,q1],[q2,f,q2]], [[q1,x],[q2,y]], q1, (x* (f*x)**) + (y* (f*y)**)).
```

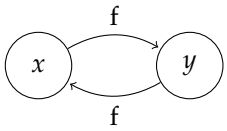
In this call our states are given by `[q1, q2]`. We have two transitions from a node to itself, given by: `[q1, f, q1]` and `[q2, f, q2]`. We have two valuations which are given by `[q1, x]` and `[q2, y]`. We start our bisimulation in node `q1` and we test acceptance by the expression `x* (f*x)**) + (y* (f*y)**`. The result of this call will be:

```
[[q1,x* (f*x)**],[q1,x* (f*x)** + y* (f*y)**]]
true.
```

This indicates that a bisimulation was possible. The behavior of `q1` in our heap is replicated by the combination of `x* (f*x)**` and `(f*x)** + y* (f*y)**` in our automaton. This bisimulation can be verified intuitively by considering the languages that the parts of this bisimulation will accept. The node `q1` in our heap will accept strings of the form $x, xfx, xfxfx, \ldots$ The node `(f*x)** + y* (f*y)**` in our automaton will accept either x followed by `x* (f*x)**` or y followed by `y* (f*y)**`. Following the path $[x, f]$ in our automaton leads to `x* (f*x)**` which also accepts strings of the form $x, xfx, xfxfx, \ldots$. Note that the node `y* (f*y)**` was never actually generated by our program because it is not necessary to simulate the behavior of our heap. If we had chosen `q2`, which accepts strings of the form $y, yfy, \ldots$, as our starting node our program would not have generated `x* (f*x)**`.

Testing the behavior of `sat()`

To demonstrate how `sat()` handles heaps and expressions that do not match perfectly we will investigate three different calls using the following heap:



First we show that this heap is accepted by the expression `x* (f*y*f*x)** * (f*y + 1)`:

```
sat([q1,q2], [[q1,f,q2],[q2,f,q1]], [[q1,x],[q2,y]], q1, x* (f*y*f*x)** * (f*y + 1)).
[[q2,[y*f*x* (f*y*f*x)** * (f*y+1),y]],[q1,[x* (f*y*f*x)** * (f*y+1)]]]
true.
```

We see that a bisimulation is possible with the following simulations: node `q2` is simulated by the nodes `y*f*x* (f*y*f*x)** * (f*y+1)` and `y`. Node `q1` is simulated by `x* (f*y*f*x)** * (f*y+1)`.

We will now add a choice to our expression to demonstrate that the bisimulation is not affected. We substitute `f*y*f*x` in our expression with `f*y*f*x + f*z*f*x` yielding the expression `x* (f*y*f*x + f*z*f*x)** * (f*y + 1)`.

```
sat([q1,q2], [[q1,f,q2],[q2,f,q1]], [[q1,x],[q2,y]], q1, x* (f*y*f*x + f*z*f*x)** * (f*y + 1)).
[[q2,[y*f*x* (f*y*f*x+f*z*f*x)** * (f*y+1),z*f*x* (f*y*f*x+f*z*f*x)** * (f*y+1),y]],
[q1,[x* (f*y*f*x+f*z*f*x)** * (f*y+1)]]]
true ;
```

We see that the bisimulation that is produced is similar to the previous example. The only difference is that in the second example node `q2` is simulated by three automaton nodes, rather than two.

We will now remove the path `f*y*f*x` from our expression to verify that our system recognizes that no acceptable paths exist.

```
sat([q1,q2], [[q1,f,q2],[q2,f,q1]], [[q1,x],[q2,y]], q1, x* (f*z*f*x)** * (f*y + 1)).
false.
```

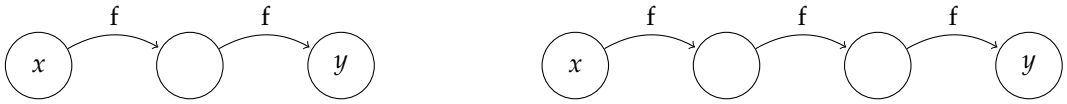
If we inspect the path Prolog followed to attempt to solve this call we see that an important failure occurs at

```
Fail: proves(y, z) ?
```

This demonstrates that our call failed because of the mismatch between `y` and `z` we introduced.

Anonymous nodes in `sat()`

As stated an important feature of heaps is that it is possible to have anonymous nodes. We will demonstrate that `sat()` is capable of testing the inclusion of heaps with anonymous nodes. We will use the expression `x+x*((f)**) *(1+y)`, which accepts $x, xf, xfy, xff, xffy, \ldots$. We will use the following heaps:



To simplify the example we omit the transition `f` from node `y` in both heaps.

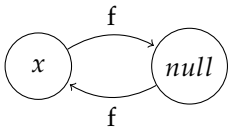
```
sat([q1,q2,q3], [[q1,f,q2],[q2,f,q3]], [[q1,x],[q3,y]], q1, x+x*((f)**) *(1+y)).
[[q3,[f** * (1+y)]],[q2,[f** * (1+y)]],[q1,[x+x*f** * (1+y)]]]
true.
```

```
sat([q1,q2,q3,q4], [[q1,f,q2],[q2,f,q3],[q3,f,q4]], [[q1,x],[q4,y]], q1, x+x*((f)**) *(1+y)).
[[q4,[f** * (1+y)]],[q3,[f** * (1+y)]],[q2,[f** * (1+y)]],[q1,[x+x*f** * (1+y)]]]
true.
```

Both heaps are accepted by our expression. Importantly, all our anonymous nodes are simulated by `f** * (1+y)` which accepts transitions over `f` with any input.

The null object in `sat()`

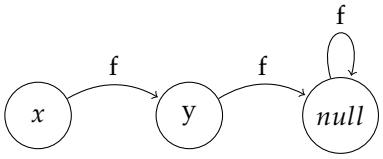
Thus far we have not dealt with the null object. Our implementation performs no validation of the heap so it expects the input to contain a valid null object if it is needed. For example, the following heap is invalid because all fields of the null object must point to itself:




```
sat([q1, q2], [[q1,f,q2], [q2,f,q1]], [[q1,x],[q2,null]], q1, x* (f*null*f*x)** * (1+f*null)).
[[q2,[null*f*x* (f*null*f*x)** * (1+f*null),null]],[q1,[x* (f*null*f*x)** * (1+f*null)]]]
true.
```

yet it is accepted by `sat()`.

If the input does not contain an invalid null object the behavior of `sat()` is as expected. For example:



tested in the following call to `sat()`:

```
sat([q1, q2, q3], [[q1,f,q2], [q2,f,q3], [q3,f,q3]], [[q1,x],[q2,y],[q3,null]], q1, x + x*f*y* (f*null)
[[q3,[null* (f*null)**]],[q2,[y* (f*null)**]],[q1,[x+x*f*y* (f*null)**]]]
true.
```

Succeeds as expected.

4.5 Loosening the interpretation of KAT-expressions

Because of the limitations we examined in 4.2 it is not trivial to craft KAT-expressions in such a way that they can accept automata. Because the purpose of this research is to construct a simpler method for describing heaps we will propose two ways of reinterpreting KAT-expressions that make them more permissive. This should make it more straightforward to write KAT-expressions that can be satisfied by a heap.

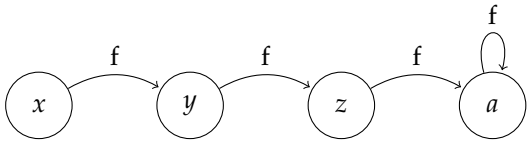
4.5.1 Accepting all substrings

Currently, all expressions we use accept all their own substrings. This is achieved using constructions such as $x + xfy$, and $x(1 + fy + fyfz)$. These constructions are necessary because any path through a KAT-automaton that contains a non-accepting state is unproductive. To remove the implicit requirement that the provided KAT-expressions accept their own substrings we will redefine the function θ.

Recall that θ was defined as $\theta(q) = out(q)$ in 2.22. We will expand this definition to also accept the boolean guard of any of the continuations from the current node as follows:

$$\theta(q) = out(q) \vee \bigvee_{(b,p) \in first(q)} b$$

We have implemented this looser interpretation in Prolog as `sat_sub()`. The following example shows how it simplifies the KAT-expression needed to accept the following heap:



The original successful call using this heap:

```
sat([q1,q2,q3,q4],[[q1,f,q2],[q2,f,q3],[q3,f,q4],[q4,f,q4]],[[q1,x],[q2,y],[q3,z],[q4,a]],q1,
        x+x*f*y+x*f*y*f*z* (f*a)**).
```

```
[[q4,[a* (f*a)**]],[q3,[z* (f*a)**]],[q2,[y,y*f*z* (f*a)**]],[q1,[x+x*f*y+x*f*y*f*z* (f*a)**]]]
true.
```

The new successful call using this heap:

```
sat_sub([q1,q2,q3,q4],[[q1,f,q2],[q2,f,q3],[q3,f,q4],[q4,f,q4]],[[q1,x],[q2,y],[q3,z],[q4,a]],q1,
        x*f*y*f*z* (f*a)**).
[[q4,[a* (f*a)**]],[q3,[z* (f*a)**]],[q2,[y*f*z* (f*a)**]],[q1,[x*f*y*f*z* (f*a)**]]]
true .
```

Note that `sat()` fails on this input:

```
sat([q1,q2,q3,q4],[[q1,f,q2],[q2,f,q3],[q3,f,q4],[q4,f,q4]],[[q1,x],[q2,y],[q3,z],[q4,a]],
        q1,x*f*y*f*z* (f*a)**).
false.
```

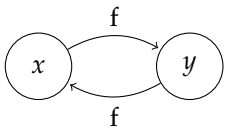
4.5.2 Accepting arbitrary continuations

In KAT-expressions, loops must be explicitly created using the Kleene star. In heaps these loops are created implicitly when a path exists from a node back to itself. Explicilty specifying these loops makes KAT-expressions more complex. To reduce the need to include loops in KAT-expressions we propose to further weaken the definition of `sat()`. by accepting all heaps that start with strings that are accepted by the automaton. In practice, we test for inclusion by $e \cdot (At \cdot \Sigma)^*$ rather than e. Programatically this is implemented by:

```
sat_tail(X, Delta, Theta, Start, Exp) :-
        sat_sub(X, Delta, Theta, Start, Exp * (a + b + c + x + y + z + f + g + h)**).
```

Note that we still make use of the simplification provided by `sat_sub()`.

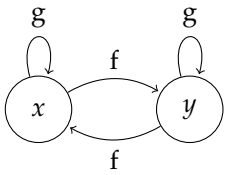
Using this procedure we can test the inclusion of the following heap and KAT-expression:




```
sat_tail([q1, q2], [[q1, f, q2],[q2,f,q1]], [[q1,x],[q2,y]], q1, x*f*y).
[[q2,[ tail]],[q1,[ tail]],[q2,[y* tail]],[q1,[x*f*y* tail]]]
true ;
```

We see that the addition of `tail` allows our expression to accept a heap which it would not accept under

`sat()` or `sat_sub()`. This does not mean that any heap will be accepted. For example the expression we used above will not accept the following heap:




```
sat_tail([q1, q2], [[q1, f, q2],[q2,f,q1], [q1, g, q1], [q2, g, q2]], [[q1,x],[q2,y]],
        q1, x*f*y).
false.
```

It is accepted by:

```
sat_tail([q1, q2], [[q1, f, q2],[q2,f,q1], [q1, g, q1], [q2, g, q2]], [[q1,x],[q2,y]], q1,
        x*f*y+x*g*x+x*f*y*g*x).
[[q1,[x* tail]],[q2,[ tail,x* tail]],[q2,[ tail]],[q1,[ tail]],
[q2,[y* tail,y*g*x* tail]],[q1,[ (x*f*y+x*g*x+x*f*y*g*x)* tail]]]
true .
```

Chapter 5

Conclusions and further work

5.0.1 Conclusions

We have shown a method for describing structures in heaps using a modification of Kleene algebra with tests. Our method simplifies the expressions needed to include a heap in two ways. Firstly, we remove the requirement that an expression include all substrings of acceptable strings. Secondly, we remove the requirement that an expression generate an infinite language. In addition to these simplifications, we provide a tool to test the inclusion of a heap in a KAT-automaton. We also refine an existing method for creating automata for KAT-expressions by eliminating incorrect transitions when a derivative is taken with regard to a non-matching boolean expression.

5.0.2 Further work

The most obvious avenue for further work is to generate a system that allows correctness proofs for programs with pointers using the methods described here. This system would probably be similar to propositional Hoare logic for sequential programs. One possible approach to this would be to translate the methods using propositional dynamic logic described in [FdBR12].

Another avenue for expanding on this work would be to devise a method to generate heaps that would be accepted by a given KAT-expression. A potential method to do this was explored during this research. The process might work as follows: 1) generate a subautomaton of the automaton belonging to the KAT-expression; 2) merge the immediate output and the boolean guard of any transition from the node into the label of the node; 3) merge nodes that can be reached with the same transition from the same node; 4) check the produced heap for consistency; 5) check whether the produced heap is accepted by the given expression.

Appendices

Appendix A

Prolog code

```
use_module(library(lists)).

:- op(100, fy, ~).
:- op(200, yf, **).

bool(a).
bool(b).
bool(c).
bool(x).
bool(y).
bool(z).
bool(1).
bool(0).
bool(null).
bool(anon).
bool(~E) :- bool(E).
bool(E1+E2) :- bool(E1), bool(E2).
bool(E1*E2) :- bool(E1), bool(E2).

nav(f).
nav(g).
nav(h).
```

```
plus(0, 0, 0) :- !.
plus(_, 1, 1) :- !.
plus(1, _, 1) :- !.
plus(E, ~E, 1) :- !.
plus(E, 0, E) :- !.
plus(0, E, E) :- !.
plus(E1, E2,E1+E2) :- !.

times(0, _, 0) :- !.
times(_, 0, 0) :- !.
times(E, ~E, 0) :- !.
times(1, E, E) :- !.
times(E, 1, E) :- !.
times(E, E, E) :- !.
times(E1, E2,E1*E2) :- !.

out([ ], [ ]).
out([H|T], Out) :- !, out(H, Out_head), out(T, Out_tail), union([Out_head], Out_tail, Out).
out(E1 + E2, Out) :- out(E1, Out_1), out(E2, Out_2), plus(Out_1, Out_2, Out), !.
out(E1 * E2, Out) :- out(E1, Out_1), out(E2, Out_2), times(Out_1, Out_2, Out), !.
out(_**, 1):- !.
out(E, E) :- bool(E), !.
out(E, 0) :- nav(E), !.

tri(_, [ ], [ ]) :- !.
tri(0, _, [ ]) :- !.
tri(B, [[Bprime, Nav]|Tail], L) :- times(B, Bprime, Bresult),
        tri(B, Tail, Lprime), union([[Bresult, Nav]], Lprime, L).

first([ ], [ ]).
first([H|T], First) :- !, first(H, First_head),
        first(T, First_tail), union(First_head, First_tail, First).
first(E1 + E2, L) :- first(E1, L1), first(E2, L2), union(L1, L2, L), !.
first(E1 * E2, L) :- first(E1, First_E1), out(E1, Out_E1),
        first(E2, First_E2), tri(Out_E1, First_E2, Tri), union(First_E1, Tri, L), !.
```

```
first(E**, L) :- first(E, L), !.
first(E, [ ]) :- bool(E).
first(E, [[1, E]]) :- nav(E).

set_times([ ], _, [ ]).
set_times(_, 0, [ ]).
set_times([1], E, [E]) :- !.
set_times([0], _, [ ]) :- !.
set_times([E1|Tail], E2, L) :- !, set_times(Tail, E2, Set_times_tail),
        times(E1, E2, Times), union([Times], Set_times_tail, L).

delta(E, L) :- first(E, First), delta(First, E, L).

delta(_, [ ], [ ]).
delta(Nav, [H|T], Delta) :- delta(Nav, H, Delta_head),
        delta(Nav, T, Delta_tail), union(Delta_head, Delta_tail, Delta).

delta([[Bool, Nav]|Tail], E, L) :- !, delta([Bool, Nav], E, Delta_head),
        delta(Tail, E, Delta_tail), union(Delta_head, Delta_tail, L).
delta([ ], _, [ ]).

delta([Bool, Nav], E1 + E2, L) :- \+is_list(Nav),
        delta([Bool, Nav], E1, Delta_1),
        delta([Bool, Nav], E2, Delta_2),
        union(Delta_1, Delta_2, L).
delta([Bool, Nav], E1 * E2, L) :- \+is_list(Nav),
        out(E1, Out), Out == 0,
        delta([Bool, Nav], E1, Delta_1),
        set_times(Delta_1, E2, Set_times),
        union(Set_times, [ ], L), !.
delta([Bool, Nav], E1 * E2, L) :- \+is_list(Nav),
        out(E1, Out), Out \== 0, substitute(E1, Bool, Bool_sub),
        delta([Bool_sub, Nav], E1, Delta_1), delta([Bool_sub, Nav], E2, Delta_2),
        set_times(Delta_1, E2, Set_times), union(Set_times, Delta_2, L).
delta([Bool, Nav], E**, L) :- \+is_list(Nav), delta([Bool, Nav], E, Delta),
```

```
        set_times(Delta, E**, Set_times), union(Set_times, [ ], L).
delta([_, Nav], E, [ ]) :- \+is_list(Nav), bool(E).
delta([Bool, Nav], E, [1]) :- \+is_list(Nav), proves(1, Bool), nav(E), Nav == E, !.
delta([_, Nav], E, [ ]) :- \+is_list(Nav), nav(E).

substitute(~A, A, 0) :- !.
substitute(A, A, 1) :- !.
substitute(A, ~E, ~E_sub) :- substitute(A, E, E_sub), !.
substitute(A + B, E, E_sub) :- replace(A, replace, E, E_sub1),
        replace(B, (~A + B), E_sub1, E_sub2), replace(replace, (A + ~B), E_sub2, E_sub), !.
substitute(A * B, E, E_sub) :- substitute(A, E, E_mid), substitute(B, E_mid, E_sub), !.
substitute(A, E1 * E2, E1_sub * E2_sub) :- substitute(A, E1, E1_sub), substitute(A, E2, E2_sub), !.
substitute(A, E1 + E2, E1_sub + E2_sub) :- substitute(A, E1, E1_sub), substitute(A, E2, E2_sub), !.
substitute(_, E, E).

replace(A, Sub, A, Sub) :- !.
replace(A, Sub, ~E, ~E_sub) :- replace(A, Sub, E, E_sub), !.
replace(A, Sub, E1 + E2, E1_sub + E2_sub) :- replace(A, Sub, E1, E1_sub), replace(A, Sub, E2, E2_sub),
replace(A, Sub, E1 * E2, E1_sub * E2_sub) :- replace(A, Sub, E1, E1_sub), replace(A, Sub, E2, E2_sub),
replace(_, _, E, E).

transitions(E, First, [Delta|Tail], Trans) :- !,
        transitions(E, First, Tail, Trans_rest), union([[E, First, Delta]], Trans_rest, Trans).
transitions(_, _, [ ], [ ]).

transitions(E, [First|Tail], Trans) :- !,
        delta([First], E, Delta), transitions(E, First, Delta, Trans_1),
        transitions(E, Tail, Trans_rest), union(Trans_1, Trans_rest, Trans).
transitions(_, [ ], [ ]).

automaton([E|Tail], Q, Delta, Theta, Done) :- automaton(E, Q_1, Delta_1, Theta_1, Done),
        automaton(Tail, Q_rest, Delta_rest, Theta_rest, [E|Done]), union(Q_1, Q_rest, Q),
        union(Delta_1, Delta_rest, Delta), union(Theta_1, Theta_rest, Theta).
automaton([ ], [ ], [ ], [ ], _).
automaton(E, [ ], [ ], [ ], Done) :- member(E, Done), !.
```

```
automaton(E, Q, Trans, Theta, Done) :- \+is_list(E),
        first(E, First), delta(First, E, Delta), transitions(E, First, Trans_1), out(E, Out),
        subtract(Delta, [E], Filtered),
        automaton(Filtered, Q_rest, Trans_rest, Theta_rest, [E|Done]),
        union([E], Delta, Q_1), union(Q_1, Q_rest, Q),
        union(Trans_1, Trans_rest, Trans), union([[E, Out]], Theta_rest, Theta).

automaton(E, Q, Trans, Theta) :- automaton(E, Q, Trans, Theta, [ ]).

proves(A, [H|T]) :- proves(A, H); proves(A, T).
proves(_, 1).
proves(_, 0) :- !, fail.

proves(A, A).
proves(~A, B) :- \+proves(A, B).
proves(A, ~B) :- \+proves(A, B).
proves(A, B + C) :- proves(A, B); proves(A, C).
proves(A, B * C) :- proves(A, B), proves(A, C).
proves(A * B, C) :- proves(A, C); proves(B, C).
proves(A + B, C) :- proves(A, C), proves(B, C).

member_or_anon([Node, Id], Label) :- member([Node, Id], Label), !.
member_or_anon([_, anon], _).

sat(X, Trans, Label, Start, E) :- bisim(X, Trans, Label, [[Start, [E]]], [ ]).

bisim(_, _, _, [ ], Done) :- print(Done).
bisim(X, Trans, Label, [[Node, E]|Todo_rest], Done) :-
        member_or_anon([Node, Id], Label), out(E, Out), proves(Id, Out),
        match(X, Trans, Label, Node, E, Todo_new),
        Done_new = [[Node, E]|Done], union(Todo_new, Todo_rest, Todo_sum),
        subtract(Todo_sum, Done_new, Todo), bisim(X, Trans, Label, Todo, Done_new).

match(_, [ ], _, _, _, [ ]) :- !.
match(X, [[Node, Nav, Target]|Trans_rest], Label, Node, E, Todo) :-
```

```
        first(E, First), member([Bool, Nav], First),
        member_or_anon([Node, Id], Label), proves(Id, Bool),
        delta([Bool, Nav], E, Delta),
        match(X, Trans_rest, Label, Node, E, Todo_rest),
        union([[Target, Delta]], Todo_rest, Todo).

match(X, [[Origin, _, _]|Trans_rest], Label, Node, E, Todo) :-
        Origin \== Node, match(X, Trans_rest, Label, Node, E, Todo).

%% ------------------------------------------------------------------------------------------

get_bool_from_nav([[Bool_head|_]|Nav_rest], Bool) :-
        get_bool_from_nav(Nav_rest, Bool_rest), union([Bool_head], Bool_rest, Bool).
get_bool_from_nav([ ], [ ]).

out_with_first(E, Out_sum) :- out(E, Out), first(E, First),
        get_bool_from_nav(First, Out_first), union(Out, Out_first, Out_sum).

bisim_no_substring(_, _, _, [ ], Done) :- print(Done).
bisim_no_substring(X, Trans, Label, [[Node, E]|Todo_rest], Done) :-
        member_or_anon([Node, Id], Label), out_with_first(E, Out), proves(Id, Out),
        match(X, Trans, Label, Node, E, Todo_new), Done_new = [[Node, E]|Done],
        union(Todo_new, Todo_rest, Todo_sum), subtract(Todo_sum, Done_new, Todo),
        bisim_no_substring(X, Trans, Label, Todo, Done_new).

sat_sub(X, Trans, Label, Start, E) :- bisim_no_substring(X, Trans, Label, [[Start, [E]]], [ ]).

%% ------------------------------------------------------------------------------------------

sat_tail(X, Trans, Label, Start, E) :-
        sat_sub(X, Trans, Label, Start, (E)* (a+b+c+x+y+z+f+g+h+null+anon+0+1)**).
```

Bibliography

[ABM12] Ricardo Almeida, Sabine Broda, and Nelma Moreira. Deciding kat and hoare logic with derivatives. 2012.

[FdBR12] Marcello Bonsangue Frank de Boer and Jurriaan Rot. Automated verification of recursive programs with pointers. *proceedings of IJCAR*, 2012.

[Joc13] Jennifer Jochems. Partial derivatives for kat expressions. *LIACS Bachelor thesis*, 2013.

[Kozoo] Dexter Kozen. On hoare logic and kleene algebra with tests. *Proc. IEEE Conf. Logic in Computer Science*, 2000.

[Pou14] Damien Pous. Symbolic algorithms for language equivalence and kleene algebra with tests, 2014.

[Sil12] Alexandra Silva. Position automata for kleene algebra with tests. *Scientific Annals of Computer Science*, 22(2):367–394, 2012.