

An Axiomatization of Arrays for Kleene Algebra with Tests

Kamal Aboul-Hosn

`kamal@cs.cornell.edu`

Department of Computer Science

Cornell University

Ithaca, NY 14853-7501, USA

Abstract. The formal analysis of programs with arrays is a notoriously difficult problem due largely to aliasing considerations. In this paper we augment the rules of Kleene algebra with tests (KAT) with rules for the equational manipulation of arrays in the style of schematic KAT. These rules capture and make explicit the essence of *subscript aliasing*, where two array accesses can be to the same element. We prove the soundness of our rules, as well as illustrate their usefulness with several examples, including a complete proof of the correctness of heapsort.

1 Introduction

Much work has been done in reasoning about programs with arrays. Arrays require more complex modeling than regular variables because of issues of *subscript aliasing*, where two array accesses can be to the same element, for example, $A(x)$ and $A(y)$ when $x = y$. Proving equivalence of programs with arrays often involves intricate read/write arguments based on program semantics or complex program transformations.

Reasoning about arrays dates back to seminal work of More [1] and Downey and Sethi [2]. Much research has also been based on early work by McCarthy on an extensional theory of arrays based on read/write operators [3]. A standard approach is to treat an array as a single variable that maps indices to values [4–6]. When an array entry is updated, say $A(i) := s$, a subsequent access $A(j)$ is treated as the program **if** $(i = j)$ **then** s **else** $A(j)$. Several other approaches of this nature are summarized in [7], where Bornat presents Hoare Logic rules for reasoning about programs with aliasing considerations.

More recently, there have been many attempts to find good theories of arrays in an effort to provide methods for the formal verification of programs with arrays. Recent work, including that of Stump et al. [8], focuses on decision procedures and NP-completeness outside the context of any formal system. Additionally, the theorem prover HOL has an applicable theory for finite maps [9].

In this paper we augment the rules of Kleene algebra with tests (KAT) with rules for the equational manipulation of arrays in the style of KAT. Introduced

in [10], KAT is an equational system for program verification that combines Kleene algebra (KA), the algebra of regular expressions, with Boolean algebra. KAT has been applied successfully in various low-level verification tasks involving communication protocols, basic safety analysis, source-to-source program transformation, concurrency control, compiler optimization, and dataflow analysis [10–16]. This system subsumes Hoare logic and is deductively complete for partial correctness over relational models [17].

Schematic KAT (SKAT), introduced in [11], is a specialization of KAT involving an augmented syntax to handle first-order constructs and restricted semantic actions. Rules for array manipulation in the context of SKAT were given in [12], but these rules were (admittedly) severely restricted; for instance, no nesting of array references in expressions was allowed. The paper [12] attempted to provide only enough structure to handle the application at hand, with no attempt to develop a more generally applicable system.

We extend the rules of [12] in two significant ways: (i) we provide commutativity and composition rules for sequences of array assignments; and (ii) we allow nested array references; that is, array references that can appear as subexpressions of array indices on both the left- and right-hand sides of assignments. The rules are *schematic* in the sense that they hold independent of the first-order interpretation.

In Section 2, we provide a brief introduction to KAT and SKAT. In Section 3, we give a set of rules for the equational manipulation of such expressions and illustrate their use with several interesting examples. These rules capture and make explicit the essence of subscript aliasing. Our main results are (i) a soundness theorem that generalizes the soundness theorem of [12] to this extended system; and (ii) a proof of the correctness of heapsort, presented in Section 5.

2 Preliminary Definitions

2.1 Kleene Algebra with Tests

Kleene algebra (KA) is the algebra of regular expressions [18, 19]. The axiomatization used here is from [20]. A *Kleene algebra* is an algebraic structure $(K, +, \cdot, *, 0, 1)$ that satisfies the following axioms:

$$\begin{array}{ll}
 (p + q) + r = p + (q + r) & (1) & (pq)r = p(qr) & (2) \\
 p + q = q + p & (3) & p1 = 1p = p & (4) \\
 p + 0 = p + p = p & (5) & 0p = p0 = 0 & (6) \\
 p(q + r) = pq + pr & (7) & (p + q)r = pr + qr & (8) \\
 1 + pp^* \leq p^* & (9) & q + pr \leq r \rightarrow p^*q \leq r & (10) \\
 1 + p^*p \leq p^* & (11) & q + rp \leq r \rightarrow qp^* \leq r & (12)
 \end{array}$$

This is a universal Horn axiomatization. Axioms (1)–(8) say that K is an *idempotent semiring* under $+$, \cdot , 0 , 1 . The adjective *idempotent* refers to (5). Axioms (9)–(12) say that p^*q is the \leq -least solution to $q + px \leq x$ and qp^* is the \leq -least solution to $q + xp \leq x$, where \leq refers to the natural partial order on K defined by $p \leq q \stackrel{\text{def}}{\iff} p + q = q$.

Standard models include the family of regular sets over a finite alphabet, the family of binary relations on a set, and the family of $n \times n$ matrices over another Kleene algebra. Other more unusual interpretations include the min,+ algebra, also known as the *tropical semiring*, used in shortest path algorithms, and models consisting of convex polyhedra used in computational geometry.

A *Kleene algebra with tests* (KAT) [10] is just a Kleene algebra with an embedded Boolean subalgebra. That is, it is a two-sorted structure $(\mathcal{K}, \mathcal{B}, +, \cdot, *, \bar{}, 0, 1)$ such that

- $(\mathcal{K}, +, \cdot, *, 0, 1)$ is a Kleene algebra,
- $(\mathcal{B}, +, \cdot, \bar{}, 0, 1)$ is a Boolean algebra, and
- $\mathcal{B} \subseteq \mathcal{K}$.

Elements of \mathcal{B} are called *tests*. The Boolean complementation operator $\bar{}$ is defined only on tests.

The axioms of Boolean algebra are purely equational. In addition to the Kleene algebra axioms above, tests satisfy the equations

$$\begin{array}{ll} BC = CB & BB = B \\ B + CD = (B + C)(B + D) & B + 1 = 1 \\ \overline{B + C} = \overline{B} \overline{C} & \overline{BC} = \overline{B} + \overline{C} \\ B + \overline{B} = 1 & B\overline{B} = 0 \\ \overline{\overline{B}} = B & \end{array}$$

2.2 Schematic KAT

Schematic KAT (SKAT) is a specialization of KAT involving an augmented syntax to handle first-order constructs and restricted semantic actions whose intended semantics coincides with the semantics of flowchart schemes over a ranked alphabet Σ [11]. Atomic propositions represent assignment operations, $x := t$, where x is a variable and t is a Σ -term.

Four identities are paramount in proofs using SKAT:

$$x := s; y := t = y := t[x/s]; x := s \quad (y \notin FV(s)) \quad (13)$$

$$x := s; y := t = x := s; y := t[x/s] \quad (x \notin FV(s)) \quad (14)$$

$$x := s; x := t = x := t[x/s] \quad (15)$$

$$\varphi[x/t]; x := t = x := t; \varphi \quad (16)$$

where x and y are distinct variables and $FV(s)$ is the set of free variables occurring in s in (13) and (14). The notation $s[x/t]$ denotes the result of substituting t for all occurrences of x in s . Here φ is an atomic first order formula. When x is not a free variable in t or in φ , we get the commutativity conditions

$$x := s; y := t = y := t; x := s \quad (y \notin FV(s), x \notin FV(t)) \quad (17)$$

$$\varphi; x := t = x := t; \varphi \quad (x \notin FV(\varphi)) \quad (18)$$

Additional axioms include:

$$x := x = 1 \quad (19)$$

$$x := s; x = s = x = s[x/s] \quad (20)$$

$$s = t; x := s = s = t; x := t \quad (21)$$

Using these axioms, one can also reason about imperative programs by translating them to propositional formulas [21]. One can translate program constructs as follows:

$$\begin{aligned} x := s &\equiv a \\ x = s &\equiv A \\ \text{if } B \text{ then } p \text{ else } q &\equiv Bp + \overline{B}q \\ \text{while } B \text{ do } p &\equiv (Bp)^* \overline{B} \end{aligned}$$

where a , is an atomic proposition and A is a Boolean test. With this translation, we can use propositional KAT to do most of the reasoning about a program independent of its meaning. We use the first order axioms only to verify premises we need at the propositional level.

3 Arrays in SKAT

Arrays have special properties that create problems when trying to reason about program equivalence. The axioms (13)-(21) do not hold without some preconditions. We want to identify the conditions under which we can apply these axioms to assignments with arrays.

Consider the statement

$$A(A(2)) := 3; A(4) := A(2)$$

We would like to use an array-equivalent version of (13) to show that

$$A(A(2)) := 3; A(4) := A(2) = A(4) := A(2); A(A(2)) := 3 \quad (22)$$

With simple variables, this sort of equivalence holds. However, in (22), if $A(2) = 2$, the two sides are not equal. The left-hand side sets both $A(2)$ and $A(4)$ to 3, while the right-hand side sets $A(2)$ to 3 and $A(4)$ to 2. The problem is that $A(2) = A(A(2))$.

One solution is to limit array indices to simple expressions that contain no array symbols, the approach taken by Barth and Kozen [12]. Let i and j be expressions containing no array symbols. For an expression e , let e_x , e_y , and e_{xy} denote $e[x/A(i)]$, $e[y/A(j)]$, and $e[x/A(i), y/A(j)]$, respectively. The following axioms hold when expressions s and t contain no array symbols and $i \neq j$:

$$A(i) := s_x; A(j) := t_{xy} = A(j) := t_y[x/s_x]; A(i) := s_x \quad (23)$$

$$A(i) := s_x; A(j) := t_{xy} = A(i) := s_x; A(j) := t_y[x/s_x] \quad (24)$$

$$A(i) := s_x; A(i) := t_x = A(i) := t[x/s_x] \quad (25)$$

$$\varphi[y/t_y]; A(j) := t_y = A(j) := t_y; \varphi \quad (26)$$

where $y \notin FV(s)$ in (23) and $x \notin FV(s)$ in (24).

These rules place some strong limitations on the programs that one can reason about, although these limitations were acceptable in the context of reasoning

in Barth and Kozen’s paper. These axioms allow no more than two array references (in most cases, only one) in a sequence of two assignment statements, which eliminates many simple program equivalences such as

$$A(3) := A(4); A(3) := A(5) = A(3) := A(5)$$

Our goal is to generalize these rules so we can have more than one array reference in a sequence of assignments and so we can allow nested array references.

In attempting to adapt (13) to arrays in a general way, we first note that an array index contains an expression that must be evaluated, which could contain another array variable. Therefore, we need to perform a substitution in that subterm as well:

$$A(i) := s; A(j) := t = A(j[A(i)/s]) := t[A(i)/s]; A(i) := s \quad (27)$$

This rule poses several questions. First of all, what is meant by $t[A(i)/s]$? We want this to mean “replace all occurrences of $A(i)$ by s in the term t .” However, this statement is somewhat ambiguous in a case such as

$$t = A(3) + A(2 + 1)$$

where i is 3. We could either replace $A(i)$ (i) *syntactically*, only substituting s for $A(3)$ in t , or (ii) *semantically*, replacing both $A(3)$ and $A(2 + 1)$. Besides being undecidable, (ii) is somewhat contrary to the sort of static analysis for which we use SKAT. Moreover, implementing these sorts of rules in a system such as KAT-ML [22] could be difficult and costly, requiring the system to perform evaluation.

However, (i) is unsound. For example,

$$\begin{aligned} A(2) := 4; A(3) := A(2) + A(1 + 1) \\ = A(3) := 4 + A(1 + 1); A(2) := 4 \end{aligned} \quad (28)$$

is not true if $A(2) \neq 4$ before execution.

Our solution is to identify the preconditions that ensure that this sort of situation does not occur. The preconditions would appear as tests in the equation to which the axiom is being applied. While it is true that establishing these preconditions is as difficult as replacing occurrences of array references semantically, it is more true to the style of SKAT, separating out reasoning that requires interpreting expressions in the underlying domain.

Let $Arr(e)$ be the set of all array references (array variable and index) that appear in the term e and let $e' = e[A(i)/s]$. We also define

$$Arrs(e, A, i, s) \stackrel{def}{=} Arr(e') - ((Arr(s) - ((Arr(e) - \{A(i)\}) \cap Arr(s)) \cap Arr(e')$$

The appropriate precondition for (27) is

$$\forall k, A(k) \in (Arrs(j, A, i, s) \cup Arrs(t, A, i, s)) \Rightarrow k \neq i$$

The condition looks complex, but what it states is relatively straightforward: any array reference that occurs in j' or t' must either not be equal to $A(i)$ or it must have been introduced when the substitution of s for $A(i)$ occurred.

For example, the transformation in (28) would be illegal, because $Arrs(A(2) + A(1 + 1), A, 2, 4)$ is $\{A(1 + 1)\}$, and $1 + 1 = 2$. However,

$$A(2) := A(2) + 1; A(3) := A(2) + 4 = A(3) := A(2) + 1 + 4; A(2) := A(2) + 1$$

would be legal, since $Arrs(A(2) + 4, A, 2, A(2) + 1)$ is the empty set.

The complexity of the condition exists because of nested array references. For example, the transformation

$$A(2) := 2; A(3) := A(A(2)) = A(3) := A(2); A(2) := 2$$

should not be legal.

We also need preconditions similar to the ones specified for (13):

$$\begin{aligned} \forall k, A(k) \in Arr(i) \cup Arr(s) &\Rightarrow k \neq j' \\ i &\neq j' \end{aligned}$$

With these preconditions, we can use the syntactic notion of replacement as we do in all other axioms. The complete set of axioms corresponding to (13)-(16) is:

$$\begin{aligned} A(i) := s; A(j) := t = A(j') := t'; A(i) := s & \quad (29) \\ \text{if } i &\neq j' \end{aligned}$$

$$\begin{aligned} \forall k, A(k) \in Arr(s) \cup Arr(i) &\Rightarrow k \neq j' \\ \forall k, A(k) \in Arrs(j) \cup Arrs(t) &\Rightarrow k \neq i \end{aligned}$$

$$\begin{aligned} A(i) := s; A(j) := t = A(i) := s; A(j') := t' & \quad (30) \\ \text{if } \forall k, A(k) \in Arr(s) \cup Arr(i) &\Rightarrow k \neq i \\ \forall k, A(k) \in Arrs(j) \cup Arrs(t) &\Rightarrow k \neq i \end{aligned}$$

$$\begin{aligned} A(i) := s; A(j) := t = A(j') := t' & \quad (31) \\ \text{if } i = j' & \\ \forall k, A(k) \in Arrs(j) \cup Arrs(t) &\Rightarrow k \neq i \end{aligned}$$

$$\begin{aligned} \varphi'; A(i) := s = A(i) := s; \varphi & \quad (32) \\ \text{if } \forall k, A(k) \in Arrs(\varphi, A, i, s) &\Rightarrow k \neq i \end{aligned}$$

We also have axioms for the interaction between assignments to array variables and to regular variables.

$$\begin{aligned} x := s; A(j) := t &= A(j[x/s]) := t[x/s]; x := s & (33) \\ \text{if } \forall k, A(k) \in \text{Arr}(s) \Rightarrow k \neq j[x/s] \end{aligned}$$

$$\begin{aligned} A(i) := s; y := t &= y := t'; A(i) := s & (34) \\ \text{if } y \notin FV(s) \cup FV(i) \\ \forall k, A(k) \in \text{Arrs}(t) \Rightarrow k \neq i \end{aligned}$$

$$\begin{aligned} x := s; A(j) := t &= x := s; A(j[x/s]) := t[x/s] & (35) \\ \text{if } x \notin FV(s) \end{aligned}$$

$$\begin{aligned} A(i) := s; y := t &= A(i) := s; y := t' & (36) \\ \text{if } \forall k, A(k) \in \text{Arr}(s) \cup \text{Arr}(i) \Rightarrow k \neq i \\ \forall k, A(k) \in \text{Arrs}(t) \Rightarrow k \neq i \end{aligned}$$

In contrast to many other treatments of arrays, we prevent aliasing through preconditions instead of using updated arrays for subsequent accesses. In approaches such as those found in [3, 6, 7], a program $A(i) := s; A(j) := t$ is translated to $A(i) := s; [A(i)/s](j) := t$, where $[A(i)/s]$ represents the array A with element i assigned to the value of s . Additionally, all occurrences of A in j and t must be replaced by $[A(i)/s]$. The replacement amounts to changing all array accesses A into the program **if** $(i = j)$ **then** s **else** $A(j)$.

Such a translation is not well suited to SKAT, where we want assignment statements to be atomic propositions. Using the if-then-else construct still requires checking all of the preconditions we have; they are captured in the test for equality of i and j . However, our precondition approach allows one to test these conditions only when doing program transformations using the axioms. Array accesses outside these transformations need not be changed at all. Since considerations of subscript aliasing primarily come up in the context of reasoning about program equivalence, it makes sense to consider aliasing through preconditions within that reasoning.

These same axioms can be extended to multidimensional arrays. Consider an array B with n indices. Each condition requiring array references to be different in the one-dimensional array case must be true in the multi-dimensional case as well. In order for two array accesses of the same array to be different, they must differ on at least one of the indices. Formally, we can state the axiom corresponding to (29) as

$$B(i_1, \dots, i_n) := s; B(j_1, \dots, j_n) := t = B(j'_1, \dots, j'_n) := t'; B(i_1 \dots i_n) := s$$

if $i \neq j'$ and:

$$\begin{aligned} \forall k_1, \dots, k_n, A(k_1, \dots, k_n) \in Arr(s) \cup \bigcup_{a=1}^n Arr(i_a) &\Rightarrow \exists \ell. 1 \leq \ell \leq n \wedge j'_\ell \neq k_\ell \\ \forall k_1, \dots, k_n, A(k_1, \dots, k_n) \in \bigcup_{a=1}^n Arrs(j_a) &\Rightarrow \exists \ell. 1 \leq \ell \leq n \wedge k'_\ell \neq i_\ell \\ \forall k_1, \dots, k_n, A(k_1, \dots, k_n) \in Arrs(t) &\Rightarrow \exists \ell. 1 \leq \ell \leq n \wedge k'_\ell \neq i_\ell \end{aligned}$$

4 Soundness of Axioms

We have proven soundness for all these rules using a technique similar to the one used in [11]. We consider interpretations over special Kripke frames called *Tarskian*, defined with respect to a first order structure D of signature Σ . States are *valuations*, assigning values in D to variables, denoted with Greek letters θ and η . For a valuation θ , $\theta[x/s]$ is the state that agrees with θ on all variables except possibly x , which takes the value s . An array variable is interpreted as a map $D \rightarrow D$, as defined in [12]. We use $\theta(A(i))$ to represent $\theta(A)(\theta(i))$.

First, we need to relate substitution in the valuation and substitution in a term. This relation corresponds to the relation between the substitution model of evaluation and the environment model of evaluation. For simple terms, this is easy:

$$\theta(t[x/s]) = \theta[x/\theta(s)](t)$$

which was shown in [11]. For arrays, we have the same difficulties of aliasing we have in the definition of our rules. The corresponding lemma for array references requires a precondition:

Lemma 1.

$$\theta(t[A(i)/s]) = \theta \left[\frac{A(\theta(i))}{\theta(s)} \right] (t)$$

if

$$\forall A(k) \in Arrs(t, A, i, s), i \neq k$$

where $\theta \left[\frac{A(i)}{s} \right]$ is the valuation that agrees with θ on all variables except possibly the array variable A , where $A(i)$ now maps to s .

Proof. The proof is a straightforward induction on t . We highlight a couple of the interesting cases. Assume $\forall A(k) \in Arrs(t, A, i, s), i \neq k$.

– $t = A(i)$: On the left-hand side, we have

$$\begin{aligned} \theta(t[A(i)/s]) &= \theta(A(i)[A(i)/s]) \\ &= \theta(s) \end{aligned}$$

From the right-hand side, we get

$$\begin{aligned} \theta \left[\frac{A(\theta(i))}{\theta(s)} \right] (t) &= \theta \left[\frac{A(\theta(i))}{\theta(s)} \right] (A(i)) \\ &= \theta(s) \end{aligned}$$

- $t = A(j)$, $j \neq i$.

$$\begin{aligned}
\theta(t[A(i)/s]) &= \theta(A(j)[A(i)/s]) \\
&= \theta(A(j[A(i)/s])) \\
&= \theta(A)(\theta(j[A(i)/s])) \\
&= \theta(A)\left(\theta\left[\frac{A(\theta(i))}{\theta(s)}\right](j)\right) \text{ by inductive hypothesis} \\
&= \theta\left[\frac{A(\theta(i))}{\theta(s)}\right](A(j))
\end{aligned}$$

The last step follows from the fact that $Arrs(j, A, i, s)$ does not contain any array index that is equal to i .

With this lemma, we can prove the soundness of (29) - (36). We show the proofs for (29) - (32), as (33) - (36) are just easier proofs that follow in the same way.

Theorem 1. $A(i) := s; A(j) := t = A(j[A(i)/s]) := t[A(i)/s]; A(i) := s$ if

$$i \neq j' \tag{37}$$

$$\forall k, A(k) \in Arr(s) \cup Arr(i) \Rightarrow k \neq j[A(i)/s] \tag{38}$$

$$\forall k, A(k) \in Arrs(j, A, i, s) \cup Arrs(t, A, i, s) \Rightarrow k \neq i \tag{39}$$

Proof. We need to show that for any Tarskian frame D ,

$$[A(i) := s; A(j) := t]_D = [A(j[A(i)/s]) := t[A(i)/s]; A(i) := s]_D$$

From the left-hand side, we have

$$\begin{aligned}
&[A(i) := s; A(j) := t]_D \\
&= [A(i) := s]_D \circ [A(j) := t]_D \\
&= \left\{ \theta, \theta \left[\frac{A(\theta(i))}{\theta(s)} \right] \mid \theta \in Val_D \right\} \circ \left\{ \eta, \eta \left[\frac{A(\eta(j))}{\eta(t)} \right] \mid \eta \in Val_D \right\} \\
&= \left\{ \theta, \theta \left[\frac{A(\theta(i))}{\theta(s)} \right] \left[\frac{A(\theta \left[\frac{A(\theta(i))}{\theta(s)} \right](j))}{\theta \left[\frac{A(\theta(i))}{\theta(s)} \right](t)} \right] \mid \theta \in Val_D \right\}
\end{aligned}$$

Now consider the right-hand side.

$$\begin{aligned}
&[A(j[A(i)/s]) := t[A(i)/s]; A(i) := s]_D \\
&= [A(j[A(i)/s]) := t[A(i)/s]]_D \circ [A(i) := s]_D \\
&= \left\{ \theta, \theta \left[\frac{A(\theta(j[A(i)/s]))}{\theta(t[A(i)/s])} \right] \mid \theta \in Val_D \right\} \circ \left\{ \eta, \eta \left[\frac{A(\eta(i))}{\eta(s)} \right] \mid \eta \in Val_D \right\} \\
&= \left\{ \theta, \theta \left[\frac{A(\theta(j[A(i)/s]))}{\theta(t[A(i)/s])} \right] \left[\frac{A(\theta \left[\frac{A(\theta(j[A(i)/s]))}{\theta(t[A(i)/s])} \right](i))}{\theta \left[\frac{A(\theta(j[A(i)/s]))}{\theta(t[A(i)/s])} \right](s)} \right] \right\}
\end{aligned}$$

where $\theta \in Val_D$.

Therefore, it suffices to show for all $\theta \in Val_D$,

$$\theta \left[\frac{A(\theta(i))}{\theta(s)} \right] \left[\frac{A(\theta \left[\frac{A(\theta(i))}{\theta(s)} \right] (j))}{\theta \left[\frac{A(\theta(i))}{\theta(s)} \right] (t)} \right] = \theta \left[\frac{A(\theta(j[A(i)/s]))}{\theta(t[A(i)/s])} \right] \left[\frac{A(\theta \left[\frac{A(\theta(j[A(i)/s]))}{\theta(t[A(i)/s])} \right] (i))}{\theta \left[\frac{A(\theta(j[A(i)/s]))}{\theta(t[A(i)/s])} \right] (s)} \right]$$

We start with the right-hand side

$$\begin{aligned} \theta \left[\frac{A(\theta(j[A(i)/s]))}{\theta(t[A(i)/s])} \right] \left[\frac{A(\theta \left[\frac{A(\theta(j[A(i)/s]))}{\theta(t[A(i)/s])} \right] (i))}{\theta \left[\frac{A(\theta(j[A(i)/s]))}{\theta(t[A(i)/s])} \right] (s)} \right] &= \theta \left[\frac{A(\theta(j[A(i)/s]))}{\theta(t[A(i)/s])} \right] \left[\frac{A(\theta(i))}{\theta(s)} \right] \quad \text{by (38)} \\ &= \theta \left[\frac{A(\theta(i))}{\theta(s)} \right] \left[\frac{A(\theta(j[A(i)/s]))}{\theta(t[A(i)/s])} \right] \quad \text{by (37)} \\ &= \theta \left[\frac{A(\theta(i))}{\theta(s)} \right] \left[\frac{A(\theta \left[\frac{A(\theta(i))}{\theta(s)} \right] (j))}{\theta \left[\frac{A(\theta(i))}{\theta(s)} \right] (t)} \right] \quad \text{by Lemma 1,} \\ &\quad (39) \end{aligned}$$

□

Theorem 2. $A(i) := s; A(j) := t = A(i) := s; A(j') := t'$ if

$$\forall k, A(k) \in Arr(s) \cup Arr(i) \Rightarrow k \neq i \quad (40)$$

$$\forall k, A(k) \in Arrs(j, A, i, s) \cup Arrs(t, A, i, s) \Rightarrow k \neq i \quad (41)$$

Proof. We need to show that for any Tarskian frame D ,

$$[A(i) := s; A(j) := t]_D = [A(i) := s; A(j[i/s]) := t[A(i)/s]]_D$$

It suffices to show for all $\theta \in Val_D$,

$$\theta \left[\frac{A(\theta(i))}{\theta(s)} \right] \left[\frac{A(\theta \left[\frac{A(\theta(i))}{\theta(s)} \right] (j))}{\theta \left[\frac{A(\theta(i))}{\theta(s)} \right] (t)} \right] = \theta \left[\frac{A(\theta(i))}{\theta(s)} \right] \left[\frac{A(\theta \left[\frac{A(\theta(i))}{\theta(s)} \right] (j[A(i)/s]))}{\theta \left[\frac{A(\theta(i))}{\theta(s)} \right] (t[A(i)/s])} \right]$$

Starting with the right-hand side

$$\begin{aligned} &\theta \left[\frac{A(\theta(i))}{\theta(s)} \right] \left[\frac{A(\theta \left[\frac{A(\theta(i))}{\theta(s)} \right] (j[A(i)/s]))}{\theta \left[\frac{A(\theta(i))}{\theta(s)} \right] (t[A(i)/s])} \right] \\ &= \theta \left[\frac{A(\theta(i))}{\theta(s)} \right] \left[\frac{A(\theta \left[\frac{A(\theta(i))}{\theta(s)} \right] \left[\frac{A(\theta \left[\frac{A(\theta(i))}{\theta(s)} \right] (i))}{\theta \left[\frac{A(\theta(i))}{\theta(s)} \right] (s)} \right] (j))}{\theta \left[\frac{A(\theta(i))}{\theta(s)} \right] \left[\frac{A(\theta \left[\frac{A(\theta(i))}{\theta(s)} \right] (i))}{\theta \left[\frac{A(\theta(i))}{\theta(s)} \right] (s)} \right] (t)} \right] \quad \text{by Lemma 1, (41)} \\ &= \theta \left[\frac{A(\theta(i))}{\theta(s)} \right] \left[\frac{A(\theta \left[\frac{A(\theta(i))}{\theta(s)} \right] \left[\frac{A(\theta(i))}{\theta(s)} \right] (j))}{\theta \left[\frac{A(\theta(i))}{\theta(s)} \right] \left[\frac{A(\theta(i))}{\theta(s)} \right] (t)} \right] \quad \text{by (40)} \\ &= \theta \left[\frac{A(\theta(i))}{\theta(s)} \right] \left[\frac{A(\theta \left[\frac{A(\theta(i))}{\theta(s)} \right] (j))}{\theta \left[\frac{A(\theta(i))}{\theta(s)} \right] (t)} \right] \quad \text{by (40)} \end{aligned}$$

□

Theorem 3. $A(i) := s; A(j) := t = A(j') := t'$ if

$$i = j' \quad (42)$$

$$\forall k, A(k) \in Arrs(j, A, i, s) \cup Arrs(t, A, i, s) \Rightarrow k \neq i \quad (43)$$

Proof. We need to show that for any Tarskian frame D ,

$$[A(i) := s; A(j) := t]_D = [A(j[i/s]) := t[A(i)/s]]_D$$

It suffices to show for all $\theta \in Val_D$,

$$\theta \left[\frac{A(\theta(i))}{\theta(s)} \right] \left[\frac{A(\theta \left[\frac{A(\theta(i))}{\theta(s)} \right] (j))}{\theta \left[\frac{A(\theta(i))}{\theta(s)} \right] (t)} \right] = \theta \left[\frac{A(\theta(j[A(i)/s]))}{\theta(t[A(i)/s])} \right]$$

Starting from the left-hand side

$$\begin{aligned} \theta \left[\frac{A(\theta(i))}{\theta(s)} \right] \left[\frac{A(\theta \left[\frac{A(\theta(i))}{\theta(s)} \right] (j))}{\theta \left[\frac{A(\theta(i))}{\theta(s)} \right] (t)} \right] &= \theta \left[\frac{A(\theta(i))}{\theta(s)} \right] \left[\frac{A(\theta(j[A(i)/s]))}{\theta(t[A(i)/s])} \right] \text{ by Lemma 1, (43)} \\ &= \theta \left[\frac{A(\theta(j[A(i)/s]))}{\theta(t[A(i)/s])} \right] \text{ by (42), (43)} \end{aligned}$$

□

Theorem 4. $\varphi'; A(i) := s = A(i) := s; \varphi$ if

$$\forall k, A(k) \in Arrs(\varphi, A, i, s) \Rightarrow k \neq i \quad (44)$$

Proof. From the left-hand side, we have

$$\begin{aligned} [\varphi[A(i)/s]; A(i) := s]_D &= [\varphi[A(i)/s]]_D \circ [A(i) := s]_D \\ &= \{(\theta, \theta) \mid \theta \in Val_D, \theta \models \varphi[A(i)/s]\} \circ \{(\theta, \theta[A(i)/s]) \mid \theta \in Val_D\} \\ &= \{\theta, \theta[A(i)/s] \mid \theta \in Val_D, \theta \models \varphi[A(i)/s]\} \end{aligned}$$

From the right-hand side, we have

$$\begin{aligned} [A(i) := s; \varphi]_D &= [A(i) := s]_D \circ [\varphi]_D \\ &= \{(\theta, \theta[A(i)/s]) \mid \theta \in Val_D\} \circ \{(\theta, \theta) \mid \theta \in Val_D, \theta \models \varphi\} \\ &= \{\theta, \theta[A(i)/s] \mid \theta \in Val_D, \theta[A(i)/s] \models \varphi[A(i)/s]\} \end{aligned}$$

From Lemma 1, we know that $\theta \models \varphi[A(i)/s]$ iff $\theta[A(i)/s] \models \varphi$. □

With these new axioms, we can prove programs equivalent that contain arrays. In all examples, fragments of the statements that changed from one step to the next are in bold>.

The following two programs for swapping array variables are equivalent, assuming that the domain of computation is the integers, $x \neq y$, and \oplus is the bitwise xor operator.

$$\begin{array}{ll} t := A(y); & A(x) := A(x) \oplus A(y); \\ A(y) := A(x); & A(y) := A(x) \oplus A(y); \\ A(x) := t; & A(x) := A(x) \oplus A(y); \\ t := 0 & t := 0 \end{array}$$

The program on the left uses a temporary variable to perform the swap while the program on the right uses properties of xor and the domain of computation

to swap without a temporary variable. We set the variable t to 0 so that the two programs end in the same state, though we could set t to any value at the end. By (15), we know that the right-hand side is equivalent to

$$\begin{aligned} A(x) &:= A(x) \oplus A(y); A(y) := A(x) \oplus A(y); \\ A(x) &:= A(x) \oplus A(y); \mathbf{t} := \mathbf{A(x)}; \mathbf{t} := \mathbf{0} \end{aligned}$$

By (34), this is equivalent to

$$\begin{aligned} A(x) &:= A(x) \oplus A(y); A(y) := A(x) \oplus A(y); \\ \mathbf{t} &:= \mathbf{A(x)} \oplus \mathbf{A(y)}; \mathbf{A(x)} := \mathbf{A(x)} \oplus \mathbf{A(y)}; \mathbf{t} := \mathbf{0} \end{aligned}$$

We then use (35) to show that this is equal to

$$\begin{aligned} A(x) &:= A(x) \oplus A(y); A(y) := A(x) \oplus A(y); \\ \mathbf{t} &:= A(x) \oplus A(y); \mathbf{A(x)} := \mathbf{t}; \mathbf{t} := \mathbf{0} \end{aligned}$$

Using (34) and (35), this is equivalent to

$$A(x) := A(x) \oplus A(y); \mathbf{t} := \mathbf{A(y)}; \mathbf{A(y)} := \mathbf{A(x)} \oplus \mathbf{t}; A(x) := \mathbf{t}; \mathbf{t} := \mathbf{0}$$

By (33), this is equivalent to

$$\mathbf{t} := \mathbf{A(y)}; \mathbf{A(x)} := \mathbf{A(x)} \oplus \mathbf{t}; A(y) := A(x) \oplus \mathbf{t}; A(x) := \mathbf{t}; \mathbf{t} := \mathbf{0}$$

By (29), where we need the condition that $x \neq y$, commutativity of xor, and the fact that $x \oplus x \oplus y = y$, this is equal to

$$\mathbf{t} := A(y); \mathbf{A(y)} := \mathbf{A(x)}; \mathbf{A(x)} := \mathbf{A(x)} \oplus \mathbf{t}; A(x) := \mathbf{t}; \mathbf{t} := \mathbf{0}$$

Finally, by (31), we end up with the left-hand side,

$$\mathbf{t} := A(y); A(y) := A(x); \mathbf{A(x)} := \mathbf{t}; \mathbf{t} := \mathbf{0}$$

5 Proving Heapsort Correct

We can prove heapsort on an array correct using these new axioms and the axioms of SKAT to get some basic assumptions so that we can reason at the propositional level of KAT. The proof is completely formal, relying only on the axioms of KAT and some basic facts of number theory. Most proofs of this algorithm are somewhat informal, appealing to a general examination of the code. An exception is a formal proof of heapsort's correctness in Coq [23].

We adapt the algorithm given in [24, Ch. 7]. Consider the function $heapify(A, i)$, which alters the array A such that the tree rooted at index i obeys the heap property: for every node i other than the root,

$$A(par(i)) \geq A(i)$$

where

$$par(i) = \lfloor i/2 \rfloor$$

We have the following property for these operators

$$i \geq 1 \Rightarrow (i = par(j) \Rightarrow rt(i) = j \vee lt(i) = j) \quad (45)$$

which states that node i is a child of its parent, where

$$\begin{aligned} lt(i) &= 2i \\ rt(i) &= 2i + 1 \end{aligned}$$

The code for the function is as follows, where the letters to the left represent the names given to the assignments and tests at the propositional level of KAT:

```
heapify(A,root)
{
a:   i := root;
B:   while(i != size(A) + 1)
    {
b:     l := lt(i);
c:     r := rt(i);
C:     if(l <= size(A) && A(l) > A(i))
d:       lgst := l
        else
e:         lgst := i
D:       if(r <= size(A) && A(r) > A(lgst))
f:         lgst := r
E:       if(lgst != i)
        {
g:         swap(A,i,lgst);
h:         i := lgst
        }
        else
j:         i := size(A) + 1
    }
}
```

where

```
swap(A,i,j)
{
    t := A[i];
    A[i] := A[j];
    A[j] := t
}
```

The variable $size(A)$ denotes the size of the heap rooted at $A(1)$ while $length(A)$ is the size of the entire array.

We wish to prove that the heapify function does in fact create the heap property for the tree rooted at index r . First, we express the property that a tree indexed at r is a heap, except for the trees under the node i and greater:

$$H'_{A,r,i} \stackrel{\text{def}}{\Leftrightarrow} 1 \leq r < i \Rightarrow \begin{aligned} &(lt(r) \leq size(A) \Rightarrow (A(r) \geq A(lt(r)) \wedge H'_{A,lt(r),i})) \wedge \\ &(rt(r) \leq size(A) \Rightarrow (A(r) \geq A(rt(r)) \wedge H'_{A,rt(r),i})) \end{aligned}$$

Now, we can easily define what it means to be a heap rooted at node r :

$$H_{A,r} \stackrel{\text{def}}{\Leftrightarrow} H'_{A,r,size(A)}$$

We can also see that

$$\begin{aligned} H_{A,r} \Leftrightarrow r \geq 1 \Rightarrow &H_{A,lt(r)} \wedge H_{A,rt(r)} \\ &\wedge (lt(r) \leq size(A) \Rightarrow A(r) \geq A(lt(r))) \\ &\wedge (rt(r) \leq size(A) \Rightarrow A(r) \geq A(rt(r))) \end{aligned} \quad (46)$$

We also have the properties

$$\begin{aligned} H'_{A,r,lt(i)} \Leftrightarrow i \geq 1 \Rightarrow &(lt(i) \leq size(A) \Rightarrow A(i) \geq A(lt(i))) \wedge \\ &(rt(i) \leq size(A) \Rightarrow A(i) \geq A(rt(i))) \wedge \\ &H'_{A,r,i} \wedge H_{A,rt(i)} \\ H'_{A,r,rt(i)} \Leftrightarrow i \geq 1 \Rightarrow &(lt(i) \leq size(A) \Rightarrow A(i) \geq A(lt(i))) \wedge \\ &(rt(i) \leq size(A) \Rightarrow A(i) \geq A(rt(i))) \wedge \\ &H'_{A,r,i} \wedge H_{A,lt(i)} \end{aligned}$$

Finally, we define the test

$$P_{A,r,i} \stackrel{\text{def}}{\Leftrightarrow} i \geq 1 \Rightarrow \begin{aligned} <(i) \leq size(A) \Rightarrow A(par(i)) \geq A(lt(i)) \wedge \\ &rt(i) \leq size(A) \Rightarrow A(par(i)) \geq A(rt(i)) \end{aligned}$$

We wish to prove that

$$root \geq 1; H_{A,lt(root)}; H_{A,rt(root)}; heapify(A, root) = heapify(A, root); H_{A,root}$$

First, we need a couple of lemmas. We show that swapping two values in an array reverses their relation, i.e.,

Lemma 2.

$$(i \neq j); A(j) \geq A(i); swap(A, i, j) = (i \neq j); swap(A, i, j); A(i) \geq A(j)$$

Proof. We need to commute the tests at the beginning through the assignments, $i \neq j$ being moved through in order to maintain the precondition we need in

order to apply (32). Then, we use (32) to move $i \neq j$ back to the beginning. Expanding swap, we start with

$$\begin{aligned}
& (i \neq j); A(j) \geq A(i); t := A(i); A(i) := A(j); A(j) := t \\
& = \mathbf{t} := \mathbf{A}(i); \mathbf{A}(i) := \mathbf{A}(j); (i \neq j); \mathbf{A}(i) \geq \mathbf{t}; A(j) := t \\
& = t := A(i); A(i) := A(j); \mathbf{A}(j) := \mathbf{t}; (i \neq j); \mathbf{A}(i) \geq \mathbf{A}(j) \\
& = (i \neq j); \mathbf{t} := \mathbf{A}(i); \mathbf{A}(i) := \mathbf{A}(j); \mathbf{A}(j) := \mathbf{t}; A(i) \geq A(j) \\
& = (i \neq j); \mathbf{swap}(\mathbf{A}, i, j); A(i) \geq A(j)
\end{aligned}$$

□

We also want to show that swapping a node i with a node j such that $H_{A,j}$ maintains the heap property for the children of j , i.e.,

Lemma 3.

$$\begin{aligned}
& H_{A,j}; t := A(i); A(i) := A(j); A(j) := t \\
& = t := A(i); A(i) := A(j); A(j) := t; \\
& \quad (j \geq 1 \Rightarrow H_{A,lt(j)} \wedge H_{A,rt(j)} \wedge (lt(j) \leq size(A) \Rightarrow A(i) \geq A(lt(j))) \wedge \\
& \quad (rt(j) \leq size(A) \Rightarrow A(i) \geq A(rt(j))))
\end{aligned}$$

Proof. We can use (46) to break down $H_{A,j}$ into $j \geq 1 \Rightarrow H_{A,lt(j)} \wedge H_{A,rt(j)} \wedge (lt(j) \leq size(A) \Rightarrow A(i) \geq A(lt(j))) \wedge (rt(j) \leq size(A) \Rightarrow A(i) \geq A(rt(j)))$. We can commute all of the conditions to the right on the left-hand side, since we know $i < j$, and therefore all the children of j are also larger than i . The move to the right replaces occurrences of $A(j)$ with $A(i)$, which gives us the right-hand side. □

The majority of the proof is spent showing the loop invariant of the while loop in the heapify function.

Lemma 4.

$$\begin{aligned}
& (i \geq 1); P_{A,root,i}; H'_{A,root,i}; H_{A,lt(i)}; H_{A,rt(i)}; \\
& (B; b; c; (C; d + \overline{C}; e)(D; f + \overline{D})(E; g; h + \overline{E}; j))^* \\
& = (B; b; c; (C; d + \overline{C}; e)(D; f + \overline{D})(E; g; h + \overline{E}; j))^*; \\
& \quad (i \geq 1); P_{A,root,i}; H'_{A,root,i}; H_{A,lt(i)}; H_{A,rt(i)}
\end{aligned}$$

Proof. From [16], we have a theorem of KAT stating

$$C; p = p; C \Rightarrow C; p^* = p^*; C$$

Therefore, it suffices to show

$$\begin{aligned}
& (i \geq 1); P_{A,root,i}; H'_{A,root,i}; H_{A,lt(i)}; H_{A,rt(i)}; \\
& B; b; c; (C; d + \overline{C}; e)(D; f + \overline{D})(E; g; h + \overline{E}; j) \\
& = B; b; c; (C; d + \overline{C}; e)(D; f + \overline{D})(E; g; h + \overline{E}); j; \\
& \quad (i \geq 1); P_{A,root,i}; H'_{A,root,i}; H_{A,lt(i)}; H_{A,rt(i)}
\end{aligned}$$

First we commute the invariants through loop guard and the first two assignments.

$$\begin{aligned}
& \mathbf{B}; \mathbf{b}; \mathbf{c}; (i \geq 1); P_{A,root,i}; H'_{A,root,i}; H_{A,l}; H_{A,r}; \\
& (C; d + \overline{C}; e)(D; f + \overline{D})(E; g; h + \overline{E}; i) \\
& meq B; b; c; (C; d + \overline{C}; e)(D; f + \overline{D})(E; g; h + \overline{E}; i); \\
& P_{A,root,i}; H'_{A,root,i}; H_{A,lt(i)}; H_{A,rt(i)}
\end{aligned}$$

Consider the three conditionals, represented by

$$(C; d + \overline{C}; e)(D; f + \overline{D})(E; g; h + \overline{E}; j)$$

We use the distributivity axiom of KAT to expand all of the terms. Some of the new terms, such as $C; d; \overline{D}; E; g; h$, correspond to assigning 1 or r to $lgst$, but the Boolean test E expecting $lgst$ to be equal to i , which is a contradiction. These pairs of tests are of the form $E; \overline{E}$, which are 0 by an axiom of KAT. We want to prove that

$$\begin{aligned}
& (i \geq 1); P_{A,root,i}; H'_{A,root,i}; H_{A,l}; H_{A,r}; \\
& (C; d; D; f; E; g; h + C; d; \overline{D}; E; g; h + \overline{C}; e; D; f; E; g; h + \overline{C}; e; \overline{D}; \overline{E}; i) \\
& = (C; d; D; f; E; g; h + C; d; \overline{D}; E; g; h + \overline{C}; e; D; f; E; g; h + \overline{C}; e; \overline{D}; \overline{E}; j); \\
& (i \geq 1); P_{A,root,i}; H'_{A,root,i}; H_{A,lt(i)}; H_{A,rt(i)}
\end{aligned}$$

It suffices to show

$$\begin{aligned}
& (i \geq 1); P_{A,root,i}; H'_{A,root,i}; H_{A,l}; H_{A,r}; C; d; D; f; E; g; h & (47) \\
& = C; d; D; f; E; g; h; (i \geq 1); P_{A,root,i}; H'_{A,root,i}; H_{A,lt(i)}; H_{A,rt(i)}
\end{aligned}$$

$$\begin{aligned}
& (i \geq 1); P_{A,root,i}; H'_{A,root,i}; H_{A,l}; H_{A,r}; C; d; \overline{D}; E; g; h & (48) \\
& = C; d; \overline{D}; E; g; h; (i \geq 1); P_{A,root,i}; H'_{A,root,i}; H_{A,lt(i)}; H_{A,rt(i)}
\end{aligned}$$

$$\begin{aligned}
& (i \geq 1); P_{A,root,i}; H'_{A,root,i}; H_{A,l}; H_{A,r}; \overline{C}; e; D; f; E; g; h & (49) \\
& = \overline{C}; e; D; f; E; g; h; (i \geq 1); P_{A,root,i}; H'_{A,root,i}; H_{A,lt(i)}; H_{A,rt(i)}
\end{aligned}$$

$$\begin{aligned}
& (i \geq 1); P_{A,root,i}; H'_{A,root,i}; H_{A,l}; H_{A,r}; \overline{C}; e; \overline{D}; \overline{E}; i & (50) \\
& = \overline{C}; e; \overline{D}; \overline{E}; j; (i \geq 1); P_{A,root,i}; H'_{A,root,i}; H_{A,lt(i)}; H_{A,rt(i)}
\end{aligned}$$

Note that (47)-(49) are extremely similar, so we provide the proof for only one of them.

(47) From the axioms of SKAT, we know that

$$C; d; D; f; E; g; h = C; N; f; g; h$$

where

$$N = r \leq size(A) \wedge A(r) > A(l)$$

Using commutativity and congruence, it suffices to show

$$\begin{aligned} & (i \geq 1); P'_{A,root,i}; H'_{A,root,i}; H_{A,l}; H_{A,lgst}; g; h \\ & = g; h; (i \geq 1); P_{A,root,i}; H'_{A,root,i}; H_{A,lt(i)}; H_{A,rt(i)} \end{aligned}$$

where $P'_{A,root,i}$ is $P_{A,root,i}$ with $A(par(i)) \geq A(lgst)$ instead of $A(par(i)) \geq A(rt(i))$ and $A(par(i)) \geq A(l)$ instead of $A(par(i)) \geq A(lt(i))$. From Lemma 3, commuting $H_{A,lgst}$ through the swap gives us

$$\begin{aligned} & (i \geq 1); P'_{A,root,i}; H'_{A,root,i}; H_{A,l}; g; \\ & (lgst \geq 1 \Rightarrow H_{A,lt(lgst)} \wedge H_{A,rt(lgst)} \wedge \\ & (lt(lgst) \leq size(A) \Rightarrow A(i) \geq A(lt(lgst))) \wedge \\ & (rt(lgst) \leq size(A) \Rightarrow A(i) \geq A(rt(lgst)))); h \\ & = g; h; (i \geq 1); P_{A,root,i}; H'_{A,root,i}; H_{A,lt(i)}; H_{A,rt(i)} \end{aligned}$$

Since we have $i \geq 1$ and $lgst = 2i + 1$, it suffices to show

$$\begin{aligned} & (i \geq 1); P'_{A,root,i}; H'_{A,root,i}; H_{A,l}; g; H_{A,lt(lgst)}; \\ & H_{A,rt(lgst)}; (lt(lgst) \leq size(A) \Rightarrow \\ & A(i) \geq A(lt(lgst))); (rt(lgst) \leq size(A) \Rightarrow \\ & A(i) \geq A(rt(lgst))); h \\ & = g; h; (i \geq 1); P_{A,root,i}; H'_{A,root,i}; H_{A,lt(i)}; H_{A,rt(i)} \end{aligned}$$

Commuting $H_{A,lt(lgst)}$ and $H_{A,rt(lgst)}$ with h gives us $H_{A,lt(i)}$ and $H_{A,rt(i)}$ at the end, which is what we want. Therefore, it suffices to show

$$\begin{aligned} & (i \geq 1); P'_{A,root,i}; H'_{A,root,i}; H_{A,l}; g; \\ & (lt(lgst) \leq size(A) \Rightarrow A(i) \geq A(lt(lgst))); \\ & (rt(lgst) \leq size(A) \Rightarrow A(i) \geq A(rt(lgst))); h \\ & = g; h; (i \geq 1); P_{A,root,i}; H'_{A,root,i} \end{aligned}$$

We know that $lgst = rt(i)$. We can replace this fact with $par(lgst) = i$, from $i \geq 1$, (45), and the fact that $lgst \neq lt(i)$. Therefore, it is sufficient to show

$$\begin{aligned} & (i \geq 1); P'_{A,root,i}; H'_{A,root,i}; H_{A,l}; g; \\ & (lt(lgst) \leq size(A) \Rightarrow A(par(lgst)) \geq A(lt(lgst))); \\ & (rt(lgst) \leq size(A) \Rightarrow A(par(lgst)) \geq A(rt(lgst))); h \\ & = g; h; (i \geq 1); P_{A,root,i}; H'_{A,root,i} \end{aligned}$$

Commuting $(lt(lgst) \leq size(A) \Rightarrow A(par(lgst)) \geq A(lt(lgst))); (rt(lgst) \leq size(A) \Rightarrow A(par(lgst)) \geq A(rt(lgst)))$ through h gives us the conditions necessary to show $P_{A,root,i}$. Now, it suffices to show

$$(i \geq 1); P'_{A,root,i}; H'_{A,root,i}; H_{A,l}; g; h = g; h; (i \geq 1); H'_{A,root,i}$$

Commuting $H_{A,l}$ through g changes nothing about it. However, we will use the fact that $l = lt(i)$. We can move $H'_{A,root,i}$ on the right side through $i \geq 1$ and h . Expanding the definition and using the fact that $lgst = rt(i)$ before h , it is sufficient to show

$$\begin{aligned} (i \geq 1); P'_{A,root,i}; H'_{A,root,i}; \mathbf{g}; \mathbf{H}_{A,lt(i)}; h = g; (lt(i) \leq size(A) \Rightarrow A(i) \geq A(lt(i))); \\ (lgst \leq size(A) \Rightarrow A(i) \geq A(lgst)); \\ \mathbf{H}'_{A,r,i}; \mathbf{H}_{A,lt(i)}; \mathbf{h}; i \geq 1 \end{aligned}$$

When we commute $H'_{A,root,i}$ through the swap, we have the same formula, except we no longer have that $1 \leq r < i \Rightarrow (i \leq size(A) \Rightarrow A(par(i)) \geq A(i))$. However, $P'_{A,root,i}$ gives us this fact when commuted through the swap; $A(par(i)) \geq A(lgst)$ becomes $A(par(i)) \geq A(i)$. We get the other two conditionals from the tests in the conditionals, commuted through the swap.

Now, we need only show $i \geq 1$ commutes through to the end. We certainly have this, given that $i \geq 1$ at the beginning and $lgst = 2i + 1$.

(50) The test $i \geq 1$ certainly commutes on the left-hand side to the end, as it does on the right-hand, so we can prove the equality in the absence of this test:

$$\begin{aligned} P_{A,root,i}; H'_{A,root,i}; H_{A,l}; H_{A,r}; \overline{C}; e; \overline{D}; \overline{E}; j \\ = \overline{C}; e; \overline{D}; \overline{E}; j; P_{A,root,i}; H'_{A,root,i}; H_{A,lt(i)}; H_{A,rt(i)} \end{aligned}$$

Now, we commute the tests on the right-hand side through the assignment j , giving us

$$\begin{aligned} P_{A,root,i}; H'_{A,root,i}; H_{A,l}; H_{A,r}; \overline{C}; e; \overline{D}; \overline{E}; j \\ = \overline{C}; e; \overline{D}; \overline{E}; \mathbf{P}_{A,root,size(A)+1}; \mathbf{H}'_{A,root,size(A)+1}; \\ \mathbf{H}_{A,lt(size(A)+1)}; \mathbf{H}_{A,rt(size(A)+1)}; j \end{aligned}$$

$P_{A,root,size(A)+1}$, $H_{A,lt(size(A)+1)}$, and $H_{A,rt(size(A)+1)}$ are vacuously true, so it remains to be shown that

$$P_{A,root,i}; H'_{A,root,i}; H_{A,l}; H_{A,r}; \overline{C}; e; \overline{D}; \overline{E}; j = \overline{C}; e; \overline{D}; \overline{E}; H'_{A,root,size(A)+1}; j$$

From the definition of H and H' ,

$$H'_{A,root,size(A)+1} \Leftrightarrow H'_{A,root,size(A)} \Leftrightarrow H(A, root)$$

Therefore, we need to show

$$P_{A,root,i}; H'_{A,root,i}; H_{A,l}; H_{A,r}; \overline{C}; e; \overline{D}; \overline{E}; j = \overline{C}; e; \overline{D}; \overline{E}; \mathbf{H}_{A,root}; j$$

We can commute $H_{A,root}$ all the way to the left. The tests \overline{C} , \overline{D} , $H'_{A,root,i}$, $H_{A,l}$, and $H_{A,r}$ combine to give us $H_{A,root}$. Note that $P_{A,root,i}$ is certainly contained within $H_{A,root}$ as well. Therefore, we can replace the heap and parent conditions with $H_{A,root}$, which gives us the right-hand side. \square

Now, we can prove the original theorem.

Theorem 5.

$$(root \geq 1); H_{A,lt(root)}; H_{A,rt(root)}; \text{heapify}(A, root) = \text{heapify}(A, root); H_{A,root}$$

Proof. At the beginning, we certainly know that $H'_{A,root,root}$ is true, so we can add it. When we commute the four tests at the beginning through the loop, we get

$$\mathbf{a}; (\mathbf{i} \geq \mathbf{1}); \mathbf{H}_{A,lt(\mathbf{i})}; \mathbf{H}_{A,rt(\mathbf{i})}; \mathbf{H}'_{A,root,\mathbf{i}}; (B; b; c; (C; d + \bar{C}; e); (D; f + \bar{D}); (E; g; h + \bar{E}; j))^*; \bar{B}$$

We know that these are invariants of the loop, so we can move them to the end.

$$\mathbf{a}; (B; b; c; (C; d + \bar{C}; e); (D; f + \bar{D}); (E; g; h + \bar{E}; j))^*; (\mathbf{i} \geq \mathbf{1}); \mathbf{H}_{A,lt(\mathbf{i})}; \mathbf{H}_{A,rt(\mathbf{i})}; \mathbf{H}'_{A,root,\mathbf{i}}; \bar{B}$$

Combined with the condition \bar{A} , $H'_{A,root,i}$ becomes $H'_{A,root,size(A)+1}$, which is $H_{A,root}$. Additionally, $H_{A,rt(size(A)+1)}$ and $H_{A,lt(size(A)+1)}$ are trivially true. Therefore, we have reduced the left side of the theorem to

$$\mathbf{a}; (B; b; c; (C; d + \bar{C}; e); (D; f + \bar{D}); (E; g; h + \bar{E}; j))^*; \bar{A}; (\mathbf{size}(A) + \mathbf{1} \geq \mathbf{1}); \mathbf{H}_{A,root}$$

The constraint on size is certainly true, since an array cannot have a negative number of elements (which can be considered an assumption $size(A) \geq 0 = 1$). Therefore, we have

$$\mathbf{a}; (B; b; c; (C; d + \bar{C}; e); (D; f + \bar{D}); (E; g; h + \bar{E}; j))^*; \bar{B}; \mathbf{H}_{A,root}$$

which is what we wanted. \square

Now that we have properties for the *heapify* function, we can show that the function *build-heap*(*A*), which creates a heap from the array *A*, works correctly. The program is

```

build-heap(A)
{
  a:   size(A) = length(A);
  b:   root := floor(size(A)/2);
  B:   while(root >= 1)
      {
  c:       heapify(A,root);
  d:       root := root - 1
      }
}

```

We show that the invariant of the loop $(B; c; d)^*$ is $\forall j > root, H_{A,j}$. It suffices to show that it is true for one iteration of the loop, i.e.

Lemma 5.

$$(\forall j > root, H_{A,j}); B; c; d = B; c; d; (\forall j > root, H_{A,j})$$

Proof. First, we define a predicate to indicate that a node i is an ancestor of node j .

$$ch(i, j) \Leftrightarrow i = par(j) \vee ch(i, par(j))$$

We can redefine $\forall j > root, H_{A,j}$ as $\forall j > root, (ch(root, j) \Rightarrow H_{A,j}) \wedge (\neg ch(root, j) \Rightarrow H_{A,j})$. It is easy to see that for all nodes for which $root$ is not an ancestor, the condition $H_{A,j}$ holds; nothing in the heapify function refers to them and they commute through unchanged. Therefore, the left-hand side becomes

$$\begin{aligned} B; (\forall j > root, ch(root, i) \Rightarrow H_{A,j}); b; \\ (\forall j > root, \neg ch(root, j) \Rightarrow H_{A,j}); c \end{aligned} \quad (51)$$

We also know that the test $\forall j > root, (ch(root, j) \Rightarrow H_{A,j})$ is true if and only if $H_{A,lt(root)}$ and $H_{A,rt(root)}$ are true. We also know that $i \geq 1$ from B . Therefore, we know from Theorem 5 and Boolean algebra rules that (51) becomes

$$B; c; (\forall j > root, H_{A,j}; H_{A,root}); d$$

Hence, we have

$$B; c; (\forall j > root - 1, H_{A,j}); d$$

which is equivalent to

$$B; c; d; (\forall j > root; H_{A,j})$$

by (16), which is what we wanted to show. \square

Now we need to show

Theorem 6.

$$a; b; (B; c; d)^*; \overline{B} = a; b; (B; c; d)^*; \overline{B}; H_{A,1}$$

Proof. From the sequence $a; b$ at the beginning, we certainly have $\forall j > root, H_{A,j}$, since the nodes greater than $root$ are leaf nodes. Therefore, the loop invariant holds, and we have

$$a; b; (B; c; d)^*; \overline{B} = a; b; (B; c; d)^*; \overline{B}; (\forall j > root, H_{A,root})$$

We now need to show

$$a; b; (B; c; d)^*; \overline{B}; (\forall j > root, H_{A,root}) = a; b; (B; c; d)^*; \overline{B}; H_{A,1}$$

This is true from the definition of $H_{A,root}$, the fact that $root < 1$ at the end, and reflexivity. Therefore, we know that the function to build a heap has the desired result. \square

Finally, we can prove that the function heapsort works. The function is defined as:

```

    heapsort(A)
  {
a:    build-heap(A);
B:    while(size(A) != 1)
      {
b:        swap(A,1,size(A));
c:        size(A) := size(A) - 1;
d:        heapify(A,1);
      }
  }

```

We want to show

Theorem 7.

$$\text{heapsort}(A) = \text{heapsort}(A); (\forall j, k, 1 \leq j < k \leq \text{length}(A) \Rightarrow A(j) \leq A(k))$$

Proof. To prove this, we prove that the invariant of the loop is

$$\begin{aligned}
& (\text{size}(A) \leq \text{length}(A) \Rightarrow A(\text{size} + 1) \geq A(\text{size}(A))); \\
& (\forall j, k, \text{size}(A) < j < k \leq \text{length}(A) \Rightarrow A(k) \geq A(j)); H_{A,1}
\end{aligned}$$

As in all the previous cases, it suffices to show the invariant is true for one execution of the loop:

$$\begin{aligned}
& (\text{size}(A) \leq \text{length}(A) \Rightarrow A(\text{size}(A) + 1) \geq A(1)); \\
& (\forall j, k, \text{size}(A) < j < k \leq \text{length}(A) \Rightarrow A(j) \leq A(k)); H_{A,1}; B; b; c; d \\
& = B; b; c; d; (\text{size}(A) \leq \text{length}(A) \Rightarrow A(\text{size} + 1) \geq A(1)); \\
& (\forall j, k, \text{size}(A) < j < k \leq \text{length}(A) \Rightarrow A(j) \leq A(k)); H_{A,1}
\end{aligned}$$

By looking at the definition of $H_{A,1}$, we know that

$$H_{A,1} \Leftrightarrow H_{A,lt(1)} \wedge H_{A,rt(1)} \wedge A(lt(1)) \leq A(1) \wedge A(rt(1)) \leq A(1) \wedge A(1) \geq A(\text{size})$$

Replacing $H_{A,1}$ with the right-hand side of this, we can commute $A(1) \geq A(\text{size})$ through the swap, reversing their relation using Lemma 2. $\forall j, k, \text{size} < j < k \leq \text{length}(A) \Rightarrow A(k) \geq A(j)$ commutes through the swap without any changes. $\text{size}(A) \leq \text{length}(A) \Rightarrow A(\text{size} + 1) \geq A(1)$ becomes $\text{size}(A) \leq \text{length}(A) \Rightarrow A(\text{size} + 1) \geq A(\text{size})$. We can combine this with $\forall j, k, \text{size}(A) < j < k \leq \text{length}(A) \Rightarrow A(k) \geq A(j)$ to get $\forall j, k, \text{size} - 1 < j < k \leq \text{length}(A) \Rightarrow A(k) \geq A(j)$. When we commute through $c; d$, we get $\forall j, k, \text{size} < j < k \leq \text{length}(A) \Rightarrow A(k) \geq A(j)$.

Then we commute $H_{A,lt(1)}$ and $H_{A,rt(1)}$ through $b; c$. The tests are still true; one of them does not change at all, and the other loses the necessary property on the parent of $A(\text{size})$, but regains it once the value of size is decreased. Therefore, we know from Theorem 5 that $H_{A,1}$ is true at the end. Therefore, we have proven these tests to be the invariant of the loop.

Now we use the invariant to show that the array A is sorted after a call to $\text{heapsort}(A)$. After $\text{build-heap}(A)$, we know that $\text{size}(A) = \text{length}(A)$, meaning

we have $\forall j, k, \text{size}(A) < j < k \leq \text{length}(A) \Rightarrow A(k) \geq A(j)$ and $\text{size}(A) \leq \text{length}(A) \Rightarrow A(\text{size}(A) + 1) \geq A(1)$ vacuously. Our proof of the loop invariant means we can move these tests to the end of the loop. Therefore, it suffices to show

$$\begin{aligned} & \overline{B}; (\forall j, k, 1 \leq j < k \leq \text{length}(A) \Rightarrow A(j) \leq A(k)) \\ &= \overline{B}; (\forall j, k, \text{size}(A) < j < k \leq \text{length}(A) \Rightarrow A(j) \leq A(k)); \\ & \quad (\text{size}(A) \leq \text{length}(A) \Rightarrow A(\text{size}(A) + 1) \geq A(1)) \end{aligned}$$

From \overline{B} , it suffices to show

$$\begin{aligned} & \forall j, k, 1 \leq j < k \leq \text{length}(A) \Rightarrow A(j) \leq A(k) \\ &= \forall j, k, 1 < j < k \leq \text{length}(A) \Rightarrow A(j) \leq A(k); 1 \leq \text{length}(A) \Rightarrow A(2) \geq A(1) \end{aligned}$$

It is easy to see that this is true. \square

Therefore, we know that the heapsort function sorts an array A .

6 Conclusions and Future Work

We have presented an axiomatization of arrays for use with KAT. Through the use of preconditions, we are able to capture the essence of aliasing considerations and consider them only where they are needed: when reasoning about program transformation. The axiomatization presented here applies to arrays. However, we believe it could be extended to pointers, since pointer analysis suffers from many of the same complications with aliasing as arrays. Providing a framework such as KAT for reasoning about pointers could be very valuable.

We would also like to implement these axioms in KAT-ML [25]. These extensions would be helpful in proving and verifying properties about everyday programs in an easily-transferable way. Arrays being so ubiquitous in programming today makes such an extension necessary to make the system useful. Inevitably, KAT and its implementation KAT-ML could provide an apparatus for verifying properties of programs written in a variety of languages.

Acknowledgments

This work was supported in part by NSF grant CCR-0105586 and ONR Grant N00014-01-1-0968. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the US Government.

References

1. More, T.: Axioms and theorems for a theory of arrays. IBM J. Res. Dev. **17**(2) (1973) 135–175

2. Downey, P.J., Sethi, R.: Assignment commands with array references. *J. ACM* **25**(4) (1978) 652–666
3. McCarthy, J.: Towards a mathematical science of computation. In: IFIP Congress. (1962) 21–28
4. McCarthy, J., Painter, J.: Correctness of a compiler for arithmetic expressions. In Schwartz, J.T., ed.: *Proceedings Symposium in Applied Mathematics*, Vol. 19, *Mathematical Aspects of Computer Science*. American Mathematical Society, Providence, RI (1967) 33–41
5. Hoare, C.A.R., Wirth, N.: An axiomatic definition of the programming language PASCAL. *Acta Informatica* **2**(4) (1973) 335–355
6. Power, A.J., Shkaravska, O.: From comodels to coalgebras: State and arrays. *Electr. Notes Theor. Comput. Sci.* **106** (2004) 297–314
7. Bornat, R.: Proving pointer programs in Hoare logic. In: MPC '00: Proceedings of the 5th International Conference on Mathematics of Program Construction, London, UK, Springer-Verlag (2000) 102–126
8. Stump, A., Barrett, C.W., Dill, D.L., Levitt, J.R.: A decision procedure for an extensional theory of arrays. In: *Logic in Computer Science*. (2001) 29–37
9. Collins, G., Syme, D.: A theory of finite maps. In Schubert, E.T., Windley, P.J., Alves-Foss, J., eds.: *Higher Order Logic Theorem Proving and Its Applications*. Springer, Berlin, (1995) 122–137
10. Kozen, D.: Kleene algebra with tests. *Transactions on Programming Languages and Systems* **19**(3) (1997) 427–443
11. Angus, A., Kozen, D.: Kleene algebra with tests and program schematology. Technical Report 2001-1844, Computer Science Department, Cornell University (2001)
12. Barth, A., Kozen, D.: Equational verification of cache blocking in LU decomposition using Kleene algebra with tests. Technical Report 2002-1865, Computer Science Department, Cornell University (2002)
13. Cohen, E.: Lazy caching in Kleene algebra (1994) <http://citeseer.nj.nec.com/22581.html>.
14. Cohen, E.: Hypotheses in Kleene algebra. Technical Report TM-ARH-023814, Bellcore (1993)
15. Cohen, E.: Using Kleene algebra to reason about concurrency control. Technical report, Telcordia, Morristown, N.J. (1994)
16. Kozen, D., Patron, M.C.: Certification of compiler optimizations using Kleene algebra with tests. In Lloyd, J., Dahl, V., Furbach, U., Kerber, M., Lau, K.K., Palamidessi, C., Pereira, L.M., Sagiv, Y., Stuckey, P.J., eds.: *Proc. 1st Int. Conf. Computational Logic (CL2000)*. Volume 1861 of *Lecture Notes in Artificial Intelligence*, London, Springer-Verlag (2000) 568–582
17. Kozen, D.: On Hoare logic and Kleene algebra with tests. *Trans. Computational Logic* **1**(1) (2000) 60–76
18. Kleene, S.C.: Representation of events in nerve nets and finite automata. In Shannon, C.E., McCarthy, J., eds.: *Automata Studies*. Princeton University Press, Princeton, N.J. (1956) 3–41
19. Conway, J.H.: *Regular Algebra and Finite Machines*. Chapman and Hall, London (1971)
20. Kozen, D.: A completeness theorem for Kleene algebras and the algebra of regular events. *Infor. and Comput.* **110**(2) (1994) 366–390
21. Fischer, M.J., Ladner, R.E.: Propositional modal logic of programs. In: *Proc. 9th Symp. Theory of Comput.*, ACM (1977) 286–294

22. Aboul-Hosn, K., Kozen, D.: KAT-ML: An interactive theorem prover for Kleene algebra with tests. In: Proc. 4th Int. Workshop on the Implementation of Logics, University of Manchester (2003) 2–12
23. Filliâtre, J.C., Magaud, N.: Certification of sorting algorithms in the Coq system. In: Theorem Proving in Higher Order Logics: Emerging Trends. (1999)
24. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms. The MIT Electrical Engineering and Computer Science Series. MIT Press/McGraw Hill (1990)
25. Aboul-Hosn, K., Kozen, D.: KAT-ML: An interactive theorem prover for Kleene algebra with tests. *Journal of Applied Non-Classical Logics* **16**(1) (2006)