Certification of Compiler Optimizations using Kleene Algebra with Tests

Dexter Kozen[1]* and Maria-Cristina Patron[2]

[1] Computer Science Department, Cornell University, Ithaca NY 14853-7501, USA.
Email: `kozen@cs.cornell.edu`

[2] Center for Applied Mathematics, Cornell University, Ithaca NY 14853-7501, USA.
Email: `mpatron@cam.cornell.edu`

Abstract. We use Kleene algebra with tests to verify a wide assortment of common compiler optimizations, including dead code elimination, common subexpression elimination, copy propagation, loop hoisting, induction variable elimination, instruction scheduling, algebraic simplification, loop unrolling, elimination of redundant instructions, array bounds check elimination, and introduction of sentinels. In each of these cases, we give a formal equational proof of the correctness of the optimizing transformation.

1 Introduction

Kleene algebra (KA) is the algebra of regular expressions. It was first introduced by Kleene in 1956 [6] and further developed in the 1971 monograph of Conway [4]. It has reappeared in many contexts in mathematics and computer science; see [7] and references therein.

In [8], an extension of KA called Kleene algebra with tests (KAT) was introduced. This system combines programs and assertions in a simple, purely equational system. In [10] it was shown that KAT strictly subsumes propositional Hoare logic, is of no greater complexity, and is deductively complete over relational models (Hoare logic is not). Moreover, KAT requires nothing beyond the constructs of classical equational logic, in contrast to Hoare logic, which requires a specialized syntax involving partial correctness assertions.

KAT has been applied successfully in various low-level verification tasks involving communication protocols, basic safety analysis, source-to-source program transformation, and concurrency control [1–3, 8]. A useful feature of KAT in this regard is its ability to accommodate certain basic equational assumptions regarding the interaction of atomic instructions. This feature makes KAT ideal for reasoning about the correctness of low-level code transformations.

In this paper we show how KAT can be used to verify a variety of common compiler optimizations: dead code elimination, common subexpression elimination, copy propagation, loop hoisting, induction variable elimination, instruction

* Supported by National Science Foundation grant CCR-9708915.

scheduling, algebraic simplification, loop unrolling, elimination of redundant instructions, array bounds check elimination, and introduction of sentinels. In each of these cases, we give a formal, machine-verifiable equational proof of the correctness of the optimizing transformation.

The verification of compiler optimizations is more than just a theoretical exercise. We were led to these investigations by recent work in typed assembly language (TAL) [12], proof-carrying code (PCC) [13], and efficient code certification (ECC) [9]. These are systems that provide a means for an untrusted compiler to convince a trusted verifier that the object code it produces meets certain safety requirements.

PCC is the most powerful of these systems. It is quite flexible in the security policies it can express, but a significant problem is the size of certificates [14]. ECC addresses this issue by taking advantage of compiler conventions, giving a significant reduction in certificate size. In ECC, the production and verification of certificates is very efficient and invisible to both the code producer and consumer. However, these savings come only at a cost of reduced expressiveness and compiler dependence. In particular, whereas TAL and PCC deal well with optimizing transformations, ECC, being more dependent on the form of the object code produced by the compiler, is less robust with respect to code motion. To verify optimized code, ECC would require the certificate to include a concise description of the sequence of optimizing transformations that were performed, along with a machine-verifiable justification of these transformations. Such an extension might be based on the system KAT as described here.

In the last section, we discuss an interesting paradox that arises in connection with *dead variables*, those whose current value will never be used. This paradox is the source of a potentially dangerous pitfall in informal reasoning. A formal treatment in KAT helps to illuminate this pitfall.

2 Kleene Algebra and Kleene Algebra with Tests

In this section we briefly review the definitions of Kleene algebra and Kleene algebra with tests; see [7] for a more thorough introduction.

2.1 Kleene Algebra (KA)

The following axiomatization is from [7]. A Kleene algebra $(K, +, \cdot, ^*, 0, 1)$ is an idempotent semiring under $+, \cdot, 0, 1$ satisfying

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

$$1 + p^*p = p^* \tag{2}$$

$$q + pr \leq r \rightarrow p^*q \leq r \tag{3}$$

$$q + rp \leq r \rightarrow qp^* \leq r, \tag{4}$$

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

$$p \leq q \overset{\text{def}}{\Longleftrightarrow} p + q = q.$$

These axioms say essentially that * behaves like the reflexive transitive closure operator of relational algebra or the Kleene asterate operator of formal languages. The operation $+$ gives the supremum with respect to $\leq$. All the operators are monotone with respect to $\leq$.

Besides basic properties of * such as $1 \leq a^*$, $a \leq a^*$, $a^*a^* = a^*$, and $a^{**} = a^*$, we will find the following two identities particularly useful:

$$p(qp)^* = (pq)^*p \tag{5}$$
$$(p+q)^* = p^*(qp^*)^* \;=\; (p^*q)^*p^*. \tag{6}$$

These identities are called the *sliding rule* and the *denesting rule*, respectively. In addition, the following result will prove useful:

Lemma 1. *In any Kleene algebra, $xy = xyx \rightarrow xy^* = x(yx)^*$.*

Proof. We show independently that

$$xy \leq xyx \rightarrow xy^* \leq x(yx)^* \tag{7}$$
$$xyx \leq xy \rightarrow x(yx)^* \leq xy^*. \tag{8}$$

To show (7), by (4) it is enough to show $xy \leq xyx \rightarrow x + x(yx)^*y \leq x(yx)^*$. Reasoning under the assumption $xy \leq xyx$, we have

$$\begin{aligned}
x + x(yx)^*y &= x + (xy)^*xy && \text{by the sliding rule (5)}\\
&\leq x + (xy)^*xyx && \text{by the assumption } xy \leq xyx\\
&= (1 + (xy)^*xy)x && \text{distributivity}\\
&= (xy)^*x && \text{by (2)}\\
&= x(yx)^* && \text{by the sliding rule (5).}
\end{aligned}$$

For (8), reasoning under the assumption $xyx \leq xy$, we have by distributivity and (1) that $x + xyxy^* \leq x + xyy^* = x(1 + yy^*) = xy^*$, thus by (3), $(xy)^*x \leq xy^*$. The right-hand side of (8) then follows from the sliding rule (5).

2.2 Kleene Algebra with Tests (KAT)

A Kleene algebra with tests is a Kleene algebra with an embedded Boolean subalgebra. Formally, it is a two-sorted structure $(K,\ B,\ +,\ \cdot,\ ^*,\ ^-,\ 0,\ 1)$ such that

- $(K,\ +,\ \cdot,\ ^*,\ 0,\ 1)$ is a Kleene algebra;
- $(B,\ +,\ \cdot,\ ^-,\ 0,\ 1)$ is a Boolean algebra; and
- $B \subseteq K$.

The Boolean complementation operator $^-$ is defined only on B.

The elements of B are called *tests*. We will denote arbitrary elements of K by the letters $p, q, r, s, t, u, v, \ldots$ and tests by $a, b, c, d, \ldots$.

When applied to arbitrary elements of K, the operators $+, \cdot, 0, 1$ refer to nondeterministic choice, composition, fail and skip, respectively. Applied to tests, they take on the additional meaning of Boolean disjunction, conjunction, falsity and truth, respectively. These two usages do not conflict—for example, sequentially testing b and c is the same as testing their conjunction bc—and their coexistence permits considerable economy of expression.

For applications in program verification, the standard interpretation would be a KA of binary relations on a set and the Boolean algebra of subsets of the identity relation. One can also consider trace models in which the Kleene elements are sets of traces (sequences of states) and the boolean elements are sets of states (traces of length 0).

The encoding of the while program constructs is as in Propositional Dynamic Logic [5]:

$$\begin{aligned} p\,;\,q &\overset{\text{def}}{=} pq \\ \textbf{if } b \textbf{ then } p \textbf{ else } q &\overset{\text{def}}{=} bp + \bar{b}q \\ \textbf{while } b \textbf{ do } p &\overset{\text{def}}{=} (bp)^*\bar{b}. \end{aligned}$$

The following result, also observed in [8], follows directly from Lemma 1. Intuitively, if the execution of the program q does not affect the value of the test b, then neither does q^*.

Lemma 2. *In any* KAT*, if $bq = qb$, then $bq^* = (bq)^*b = q^*b = b(qb)^*$.*

Proof. If $bq = qb$, then by Boolean algebra $bq = bbq = bqb$, thus $bq^* = b(qb)^*$ by Lemma 1. The other equations follow from the sliding rule (5) and symmetry.

2.3 KAT and Hoare Logic

Hoare logic is a system for deriving partial correctness properties of compound programs compositionally from properties of their constituent parts. Traditionally, these properties are expressed by *partial correctness assertions* (PCAs) of the form $\{b\}\,p\,\{c\}$, where b and c are assertions in the underlying assertion language and p is a program. Intuitively, the PCA $\{b\}p\{c\}$ says that if the property b holds at the start of execution of p, and if p halts, then c must be true in the halting state.

As mentioned in the introduction, KAT subsumes Hoare logic [10]. The PCA $\{b\}\,p\,\{c\}$ is expressed $bp\bar{c} = 0$, or equivalently, $bp = bpc$. Intuitively, $bp\bar{c} = 0$ says that there is no halting computation of p satisfying precondition b and postcondition $\bar{c}$, and $bp = bpc$ says that testing c after executing p with precondition b is always redundant.

In traditional Hoare logic, atomic programs are assignments $x := e$ and the only atomic assumption is the *assignment rule* $\{P[x/e]\}\,x := e\,\{P\}$. Hoare logic operates by deriving PCAs involving compound programs inductively, using the assignment rule as an axiom. The operation of KAT is analogous, except that the assumptions and conclusions are equations between programs, and the form of the assumptions can be more general. Theorems of KAT are universally quantified Horn formulas of the form $(\bigwedge_i p_i = q_i) \to p = q$. In our applications below, the $p_i = q_i$ are typically premises that involve atomic instructions and tests that are immediately self-evident, and the conclusion $p = q$ is the equivalence of the unoptimized and optimized code fragments.

In our optimization examples, there are certain kinds of premises that occur frequently. For example, we often need to know that two atomic instructions that do not affect each other can occur in either order. This would be expressed in KAT by a commutativity condition of the form $pq = qp$. We would take this assertion as a premise on the left-hand side of the Horn formula above. Another common example is the fact that after loading a register with a value, that register contains that value. This is expressed by an equation of the form $p = pa$, where p is the load instruction and a is the assertion that the register contains the value. This assertion allows us to introduce new assertions into an annotated program and delete them when they are no longer needed. As a final example, the fact that if a register already contains a value, then there is no need to load it again would be encoded as an equation of the form $ap = a$. This premise allows us to delete redundant instructions.

We use such atomic premises extensively in the derivations of Section 3. In all cases the truth of the premise is directly evident. Moreover, it has been observed that in the decision procedure for KAT, premises of the form $p = 0$ can be eliminated without loss of efficiency [1, 11].

3 Verifying Optimizations in KAT

In this section we consider several examples of common compiler optimizations and show how they can be encoded and verified in KAT. In each case, we give the program fragments before and after the optimizations, their translations into the language of KAT, and an equational proof that the two fragments are equivalent.

3.1 Dead Code Elimination

Dead code elimination is a code transformation that removes unreachable instructions. Let us start with a very simple example. Consider the program p ; **if** a **then** q. This is expressed in KAT as $p(aq + \bar{a})$. The $\bar{a}$ in this expression represents the implicit **else** clause. Suppose we know that the test a is always false after the execution of p. This would imply that the test of the **if** statement is false in the program above, so q would never be executed. We could remove it to obtain the optimized fragment p.

The assumption that the test a is always false after the execution of p is expressed in KAT by the identity $p = p\bar{a}$, or equivalently $pa = 0$. Intuitively, immediately after the execution of p, we must always be in a state in which $\bar{a}$ holds. In this case, executing the guard $\bar{a}$ after p is always redundant; equivalently, executing the guard a after p aborts the computation.

Reasoning in KAT under the assumption $p = p\bar{a}$, we have

$$p(aq + \bar{a}) = p\bar{a}(aq + \bar{a}) = p\bar{a}aq + p\bar{a}\bar{a} = p0q + p\bar{a} = 0 + p\bar{a} = p\bar{a} = p.$$

Thus the KAT expressions representing the two program fragments are equal.

For the case of a **while** loop, consider the fragment p ; **while** a **do** q, which is encoded in KAT as the expression $p(aq)^*\bar{a}$. Again, suppose that the test a is always false after the execution of p; that is, $p\bar{a} = p$. This means that the **while** loop will never be executed, and we should again be able to obtain the optimized fragment p.

As above, reasoning in KAT under the assumption $p = p\bar{a}$, we have

$$\begin{aligned} p(aq)^*\bar{a} &= p\bar{a}(aq)^*\bar{a} = p\bar{a}(1 + aq(aq)^*)\bar{a} = p\bar{a}\bar{a} + p\bar{a}aq(aq)^*\bar{a} \\ &= p\bar{a} + p0q(aq)^*\bar{a} = p\bar{a} + 0 = p\bar{a} = p. \end{aligned}$$

Both of these cases give examples of how assumptions about atomic programs and tests (here $p = p\bar{a}$) are used to derive the equivalence of the unoptimized and optimized programs. We have essentially given purely equational proofs of the universal Horn formulas $p = p\bar{a} \rightarrow p(aq + \bar{a}) = p$ and $p = p\bar{a} \rightarrow p(aq)^*\bar{a} = p$.

3.2 Common Subexpression Elimination

Common subexpression elimination is a code transformation that avoids redundant evaluation of the same expression by using the result of the first computation. For example, consider the program fragment $i := \mathit{expr}\ ;\ j := \mathit{expr}$, where *expr* is an expression not containing i. We wish to show that this can be replaced by $i := \mathit{expr}\ ;\ j := i$.

Consider the following programs and tests:

$$\begin{aligned} p &\overset{\text{def}}{=} i := \mathit{expr} & w &\overset{\text{def}}{=} \text{make } j \text{ undefined} \\ q &\overset{\text{def}}{=} j := \mathit{expr} & a &\overset{\text{def}}{\iff} i = \mathit{expr} \\ r &\overset{\text{def}}{=} j := i & b &\overset{\text{def}}{\iff} i = j. \end{aligned}$$

We wish to prove that $pq = pr$. We can postulate the following premises:

$1p = 1pa$	atomic PCA $\{\mathit{expr} = \mathit{expr}\}\ i := \mathit{expr}\ \{i = \mathit{expr}\}$
$aq = aqb$	atomic PCA $\{i = \mathit{expr}\}\ j := \mathit{expr}\ \{i = j\}$
$br = b$	there is no need to assign $j := i$ if $i = j$ already
$r = wr$	j is dead immediately before the assignment $j := i$
$qw = w$	an assignment to a dead variable is redundant.

The first two of these are both instances of the Hoare assignment rule. Under these premises, we can reason equationally as follows:

$$\begin{aligned} pq &= 1pq = 1paq = 1paqb = 1paqbr = 1paqr \\ &= 1pqr = pqr = pqwr = pwr = pr. \end{aligned}$$

3.3 Copy Propagation

Copy propagation is a code transformation that eliminates an assignment of the form $j := i$ and replaces all further references to j by references to i. For example, consider the program fragment

$$i := \mathit{expr}\,;\; j := \mathit{expr}\,;\; k := 4 * j + 2$$

where i and j do not occur in expr. By common subexpression elimination (Section 3.2), the second assignment can be replaced by $j := i$.

First we argue that we can replace the last assignment by $k := 4 * i + 2$. Letting p, q, r, and s denote the assignments $i := \mathit{expr}$, $j := i$, $k := 4 * j + 2$, and $k := 4 * i + 2$ respectively, we wish to show that $pqr = pqs$. It suffices to show that $qr = qs$. Consider the program and tests

$$\begin{aligned} a &\overset{\text{def}}{\iff} 4 * j + 2 = 4 * i + 2 \\ b &\overset{\text{def}}{\iff} k = 4 * i + 2 \\ w &\overset{\text{def}}{=} \text{ make } k \text{ undefined.} \end{aligned}$$

As above, we postulate the following premises:

$1q = 1qa$	atomic PCA $\{4 * i + 2 = 4 * i + 2\}\; j := i\; \{4 * j + 2 = 4 * i + 2\}$
$ar = arb$	atomic PCA $\{4 * j + 2 = 4 * i + 2\}\; k := 4 * j + 2\; \{k = 4 * i + 2\}$
$bs = b$	there is no need to assign $k := 4 * i + 2$ if $k = 4 * i + 2$ already
$s = ws$	k is dead immediately before the assignment $k := 4 * i + 2$
$rw = w$	an assignment to a dead variable is redundant.

The first two of these are instances of the Hoare assignment rule. Using these assumptions, we can reason as follows:

$$\begin{aligned} qr &= 1qr = 1qar = 1qarb = 1qarbs = 1qars \\ &= 1qrs = qrs = qrws = qws = qs. \end{aligned}$$

Moreover, if we know that j is a dead variable, we can optimize further by removing the assignment to j, obtaining the optimized code ps. Letting $v \overset{\text{def}}{=}$ "make j undefined", we wish to show that $pqsv = psv$. We have $sv = vs$, since j does not occur in s, and $qv = v$, since if j is dead, the assignment is redundant. This allows us to conclude $pqsv = pqvs = pvs = psv$.

3.4 Loop Hoisting

Loop hoisting is a transformation that involves moving code out of loops. It can take one of two forms: in the first form, an expression whose value does not depend on the number of times through the loop need not be evaluated inside the loop, but can be evaluated once before the first execution of the body of the loop. In the second, an expression whose value is not used anywhere inside the loop need not be evaluated inside the loop, but can be evaluated once after the loop.

As an example of the first type of transformation, consider the following program fragment:

```
$sum := 1\,;$                         $p$
while $1 \le i \le n$ do {
   $sum := sum + i * expr\,;$         $q$
   $i := i + 1\,;$                    $s$
}
```

where *expr* is an expression not containing i or *sum*. Let k be a new variable. This fragment is equivalent to the fragment

```
$sum := 1\,;$                         $p$
$k := expr\,;$                        $u$
while $1 \le i \le n$ do {
   $sum := sum + i * k\,;$            $r$
   $i := i + 1\,;$                    $s$
}
```

Formally, "k is a new variable" is captured by saying that k does not appear in any expression in the first fragment and that k can be made undefined immediately after the execution of the fragment.

Define the program and tests

$$\begin{aligned} a &\overset{\text{def}}{\iff} 1 \le i \le n \\ b &\overset{\text{def}}{\iff} k = expr \\ w &\overset{\text{def}}{=} \text{ make } k \text{ undefined.} \end{aligned}$$

We would like to show $p(aqs)^*\bar{a}w = pu(ars)^*\bar{a}w$. Postulating the assumptions

$u = ub$	$k = expr$ after $k := expr$, since k does not occur in $expr$
$b = bu$	if $k = expr$ already, no need to assign $k := expr$
$bq = qb$	since sum does not occur in $expr$
$bs = sb$	since i does not occur in $expr$
$br = rb$	since sum does not occur in $expr$,

using Lemma 2 and copy propagation (Section 3.3) we can argue as follows:

$$\begin{aligned} pu(ars)^* &= pub(ars)^* = pub(abrs)^* = pub(aburs)^* \\ &= pub(abqs)^* = pub(aqs)^* = pu(aqs)^*. \end{aligned}$$

Now since w commutes with a, $\bar{a}$, q, and s, we have by Lemma 2 that $w(aqs)^* = (aqs)^*w$. Also, $uw = w$ since there is no need to assign to a dead variable. Thus

$$pu(aqs)^*\bar{a}w = puw(aqs)^*\bar{a} = pw(aqs)^*\bar{a} = p(aqs)^*\bar{a}w.$$

In conclusion, $pu(ars)^*\bar{a}w = p(aqs)^*\bar{a}w$, which is what we had to prove.

As an example of the second type of transformation, consider the following program in which the computation p inside the loop and the test a do not use i:

```
while $a$ do {                  
    $i := r$ ;              $u$
    $p$ ;                   $p$
    $r := r + 1$ ;          $q$
}
$i := r$ ;                  $u$
```

Since i is assigned a different expression each time the loop is executed, the previous example does not apply. Nevertheless, since i is not used in the rest of the loop, we still obtain the optimized code:

```
while $a$ do {
    $p$ ;                   $p$
    $r := r + 1$ ;          $q$
}
$i := r$ ;                  $u$
```

We would like to prove $(aupq)^*\bar{a}u = (apq)^*\bar{a}u$. Defining the atomic program $w \overset{\text{def}}{\Longleftrightarrow}$ "make i undefined", we have the following postulates:

$u = wu$	i is dead just before the assignment $i := r$
$wpq = pqw$	p and q do not refer to i
$wa = aw$	a does not refer to i
$w\bar{a} = \bar{a}w$	a does not refer to i
$uw = w$	an assignment to a dead variable is redundant.

Reasoning under these assumptions using the sliding rule (5) and Lemma 2,

$$\begin{aligned}(aupq)^*\bar{a}u &= (awupq)^*\bar{a}wu = (waupq)^*w\bar{a}u = w(aupqw)^*\bar{a}u = w(auwpq)^*\bar{a}u \\ &= w(awpq)^*\bar{a}u = (apq)^*w\bar{a}u = (apq)^*\bar{a}wu = (apq)^*\bar{a}u.\end{aligned}$$

3.5 Induction Variable Elimination

This is a loop optimization that replaces multiplicative operations inside the loop with less expensive additive ones. This type of optimization might arise in matrix algorithms. For example, consider the program

```
$i := init$ ;                   $u$
$j := i * expr_2$ ;             $q$
while $a$ do {
    $i := i + expr_1$ ;         $p$
    $j := i * expr_2$ ;         $q$
}
```

where i and j do not occur in $expr_1$ and $expr_2$. Note that whenever i is increased by $expr_1$, j is increased by $expr_1 * expr_2$. The optimized code is

```
$i := init$ ;                          $u$
$j := i * expr_2$ ;                    $q$
while $a$ do {
   $i := i + expr_1$ ;                 $p$
   $j := j + expr_1 * expr_2$ ;        $r$
}
```

Using the transformation of Section 3.4, we can further optimize to obtain

```
$i := init$ ;
$j := i * expr_2$ ;
$m := expr_1$ ;
$n := expr_1 * expr_2$ ;
while $a$ do {
   $i := i + m$ ;
   $j := j + n$ ;
}
```

To establish the equivalence of the first two programs, we need to prove

$$uq(apq)^*\bar{a} = uq(apr)^*\bar{a}.$$

It suffices to prove $q(apq)^* = q(apr)^*$. Consider the tests

$$\begin{aligned} b &\overset{\text{def}}{\iff} j = i * expr_2, \\ b' &\overset{\text{def}}{\iff} j + expr_1 * expr_2 = (i + expr_1) * expr_2 \\ c &\overset{\text{def}}{\iff} j + expr_1 * expr_2 = i * expr_2 \end{aligned}$$

We have the assumptions $q = qb$; $b = bq$; $b = b'$ from basic number-theoretic reasoning; $cr = crb$, an instance of the Hoare assignment rule; and $bp = bpc$, which follows from $b = b'$ and the instance $\{b'\}\,p\,\{c\}$ of the Hoare assignment rule. In addition, we have $cq = cr$, which is an instance of the property that if two expressions have the same value, then the assignment of either expression to the variable j has the same effect. This would hold even if j occurred in both expressions. Here, j does not occur in the expression $i * expr_2$, and using $w \overset{\text{def}}{\iff}$ "make j undefined" along with the premises $wq = q$ and $rw = w$, $cq = cr$ can be proved by

$$cr = crb = crbq = crbwq = crwq = cwq = cq.$$

The property $cq = cr$ holds even in the more general case in which j can occur in both expressions. We do not know how to prove this in Hoare logic or Kleene algebra from more primitive assumptions without introducing new symbols into the underlying programming or assertion language. However, we are content to take $cq = cr$ as a primitive assumption.

We have $bpq = bpcq = bpcqb = bpcrb = bpcr = bpr$. Since $bpq = bpqb$, it follows that $bapq = bapqb$ and $bapr = baprb$. Using the sliding rule (5) and Lemma 1, we then have

$$q(apq)^* = qb(apqb)^* = q(bapq)^*b = q(bapr)^*b = qb(aprb)^* = q(apr)^*.$$

3.6 Instruction Scheduling

Unrelated instructions can be reordered so as to maximize the throughput of a processor pipeline. For example, $p\,;q$ and $q\,;p$ are equivalent if there is no dependency between the instructions p and q. The nondependency assumption is expressed in KAT by the equation $pq = qp$. These assumptions can be used to reorder instructions arbitrarily as long as no dependencies are violated.

3.7 Algebraic Simplification

This transformation eliminates statements corresponding to trivial algebraic identities, which occasionally arise due to constant propagation and other previous transformations. For example, any assignment of the form $i := i + 0$ or $i := i * 1$ can be eliminated. This is simply an application of an assumption of the form $ap = a$ and the Kleene algebra axiom $1q = q$.

3.8 Loop Unrolling

Sometimes it is possible to reduce the number of tests and jumps executed in a loop by unrolling the loop. We can unroll the loop **while** a **do** p once to obtain **while** a **do** $\{p\,;$ **if** a **then** $p\}$. We have to prove $(ap)^*\bar{a} = (ap(ap+\bar{a}))^*\bar{a}$. The following lemma of pure KAT captures the essence of this transformation.

Lemma 3. *In any Kleene algebra,* $u^* = (1+u)(uu)^*$.

Proof. For the direction $\geq$, by monotonicity, distributivity, idempotence, denesting (6), and the basic properties of *, we have

$$(1+u)(uu)^* = (uu)^* + u(uu)^* \leq u^*(uu^*)^* = (u+u)^* = u^*.$$

For the direction $\leq$, by (3) it is enough to prove $1+u(1+u)(uu)^* \leq (1+u)(uu)^*$. By (1) and distributivity, we have

$$1 + u(1+u)(uu)^* = u(uu)^* + 1 + uu(uu)^* = u(uu)^* + (uu)^* = (1+u)(uu)^*.$$

We can now prove the equivalence of the two programs using sliding (5), denesting (6), the basic axioms, and Lemma 3.

$$\begin{aligned}
&(ap(ap+\bar{a}))^*\bar{a} = (apap + ap\bar{a})^*\bar{a} = ((ap\bar{a})^*apap)^*(ap\bar{a})^*\bar{a}\\
&= ((1+(ap\bar{a})^*ap\bar{a})apap)^*(ap\bar{a})^*\bar{a} = (apap + (ap\bar{a})^*ap\bar{a}apap)^*(ap\bar{a})^*\bar{a}\\
&= (apap)^*(ap\bar{a})^*\bar{a} = (apap)^*(1+ap\bar{a}(ap\bar{a})^*)\bar{a}\\
&= (apap)^*(1+ap\bar{a}+ap\bar{a}ap\bar{a}(ap\bar{a})^*)\bar{a} = (apap)^*(1+ap\bar{a})\bar{a}\\
&= (apap)^*\bar{a} + ap(apap)^*\bar{a} = (1+ap)(apap)^*\bar{a} = (ap)^*\bar{a}.
\end{aligned}$$

3.9 Redundant Loads and Stores

In the instruction sequence **load** r, i ; **store** r, i, the store instruction is redundant, since the first ensures that the value of i is the same as the contents of register r. We obtain the optimized code **store** r, i. Letting $p \overset{\text{def}}{=}$ **load** r, i, $q \overset{\text{def}}{=}$ **store** r, i, and $a \overset{\text{def}}{\iff} r = i$, we can postulate $p = pa$, since after loading i into register r, the test $r = i$ is redundant; and $aq = a$, since storing r in i is redundant if the value is already there. Under these assumptions, we have $pq = paq = pa = p$.

3.10 Array Bounds Check Elimination

Consider the following program to initialize the elements of an array:

$$i := 0\,;\ \textbf{while}\ i < x.\mathit{length}\ \textbf{do}\ \{x[i] := e(i)\,;\ i := i + 1\}$$

A compiler has to check that array accesses fall within bounds:

```
    $i := 0$                                  $u$
$\alpha$ : **test** $i \geq x.length$
    **jtrue** $\beta$
    **compute** $e(i)$                        $p$
    **if** $i$ in bounds **then** $x[i] := e(i)$  $q$
      **else error**                          $s$
    $i := i + 1$                              $v$
    **goto** $\alpha$
$\beta$ : ...
```

The bounds check inside the loop is redundant. The optimized code is

```
    $i := 0$               $u$
$\alpha$ : **test** $i \geq x.length$
    **jtrue** $\beta$
    **compute** $e(i)$     $p$
    $x[i] := e(i)$         $q$
    $i := i + 1$           $v$
    **goto** $\alpha$
$\beta$ : ...
```

Consider also the tests

$$a \overset{\text{def}}{\iff} 0 \leq i$$

$$b \overset{\text{def}}{\iff} i < x.\mathit{length}$$

$$c \overset{\text{def}}{\iff} ab \iff i \text{ is in bounds.}$$

We have to prove $u(bp(cq + \bar{c}s)v)^*\bar{b} = u(bpqv)^*\bar{b}$. We see that if a is true at the beginning of the loop, it remains true after one iteration; that is, $a(bp(cq+\bar{c}s)v) =$

$a(bp(cq+\bar{c}s)v)a$. Reasoning under the assumptions $u = ua$, $ab = c$, $pc = cp$, and $a(bpqv) = (bpqv)a$ and using dead code elimination (Section 3.1), and Lemma 2, we have

$$\begin{aligned} u(bp(cq+\bar{c}s)v)^*\bar{b} &= ua(bp(cq+\bar{c}s)v)^*\bar{b} = ua(abp(cq+\bar{c}s)v)^*\bar{b} \\ &= ua(cp(cq+\bar{c}s)v)^*\bar{b} = ua(pc(cq+\bar{c}s)v)^*\bar{b} = ua(pcqv)^*\bar{b} \\ &= ua(abpqv)^*\bar{b} = ua(bpqv)^*\bar{b} = u(bpqv)^*\bar{b}. \end{aligned}$$

Note that KAT does *not* contain explicit machinery for number-theoretic reasoning; that is a separate issue. However, as shown in this example, it does reduce the correctness of the optimizing code transformation to a set of basic number-theoretic assumptions on atomic programs and tests that justify the transformation.

3.11 Introduction of Sentinels

Our last example is also related to arrays. Suppose we want to check if a certain element, say T, is among the elements of a nonempty array x of length n. This can be done by:

```
$i := 0$ ;                                      $p$
**while** $i < n$ and $x[i] \neq T$ **do** {
   $i := i + 1$ ;                               $q$
}
**if** $i < n$ **then** found = true ;          $t$
**else** found = false ;                        $s$
```

In order to eliminate one of the tests of the **while** loop, we introduce a *sentinel*: we extend the array x by a new element initialized with T. The optimized program is

```
$x[n] := T$ ;                                   $u$
$i := 0$ ;                                      $p$
**while** $x[i] \neq T$ **do** {
   $i := i + 1$ ;                               $q$
}
**if** $i < n$ **then** found = true ;          $t$
**else** found = false ;                        $s$
```

To prove that the two programs are equivalent, consider the tests

$$\begin{array}{ll} a \overset{\text{def}}{\iff} i < n & \quad c \overset{\text{def}}{\iff} x[n] = T \\ b \overset{\text{def}}{\iff} x[i] \neq T & \quad d \overset{\text{def}}{\iff} i \leq n. \end{array}$$

Since $x[n]$ will not be used further in the program, we can also use $w \overset{\text{def}}{\iff}$ "make $x[n]$ undefined". We want to prove

$$p(abq)^*\bar{a}\bar{b}(at+\bar{a}s)w = up(bq)^*\bar{b}(at+\bar{a}s)w. \tag{9}$$

Since $uw = w$ and u commutes with the programs p, q, s, t and the tests a and ab, we can introduce u on the left-hand side of (9) and move it to the front of the expression using Lemma 2. It therefore suffices to prove

$$up(abq)^*\bar{ab}(at + \bar{a}s)w = up(bq)^*\bar{b}(at + \bar{a}s)w.$$

Since $u = uc$, $cp = pc$, and $p = pd$, we have $up = upcd$, thus it suffices to prove

$$cd(abq)^*\bar{ab} = cd(bq)^*\bar{b}. \tag{10}$$

Now note that $cdb \le a$, or in other words $cdb = cdba$, $cq = qc$, and $aq = aqd$. Then

$$cdbq = cdbaq = cdbaqd = ccdbaqd = cdbaqcd = cdbqcd.$$

Using Lemma 1 variously with $x = cd$ and $y = abq$ and with $x = cd$ and $y = bq$, sliding (5), and the properties $cd\bar{a} \le \bar{b}$ and $cdba = cdb$, we have

$$\begin{aligned} cd(abq)^*\bar{ab} &= cd(abqcd)^*\bar{ab} = (cdabq)^*cd(\bar{a} + \bar{b}) = (cdabq)^*(cd\bar{a} + cd\bar{b}) \\ &= (cdabq)^*cd\bar{b} = (cdbq)^*cd\bar{b} = cd(bqcd)^*\bar{b} = cd(bq)^*\bar{b}. \end{aligned}$$

This proves (10).

4 The Dead Variable Paradox

We conclude with some remarks about an interesting paradox concerning dead variables (variables whose values will never be used). This paradox is the source of a potentially dangerous pitfall that can arise when reasoning informally about the liveness of variables. A formal treatment in KAT helps to illuminate this issue.

The reader will have noticed that we have made extensive use of the construct

$$w \overset{\text{def}}{=} \text{make } i \text{ undefined},$$

along with the atomic assertions $pw = w$ and $wp = p$, where p is an assignment to i of an expression not containing i, and may have wondered why we did not use the test

$$d \overset{\text{def}}{\iff} i \text{ is a dead variable}$$

and the assertions $pd = d$ and $dp = p$ instead. For example, if $p \overset{\text{def}}{=} i := 1$ and $q \overset{\text{def}}{=} j := 2$, we could postulate the atomic premises

$p = dp$	i is dead immediately before the assignment p
$qd = dq$	the assignment q does not affect i
$pd = d$	an assignment to a dead variable is redundant,

then eliminate the first assignment to i in the program $i := 1\,;\, j := 2\,;\, i := 1$ by arguing $pqp = pqdp = pdqp = dqp = qdp = qp$.

The problem is that the proposition "i is a dead variable" is not a property of the local state of the computation. It does not commute with other tests involving i, which it must do in order to be a Boolean element of a Kleene algebra with tests, and its use as a test in the context of KAT can lead to paradoxical results.

To illustrate, consider the following calculation. Defining $a \overset{\text{def}}{\iff} i = 1$, we have $p = pa$, since $i = 1$ immediately after the assignment, and $ap = a$, since the assignment is redundant if $i = 1$ already. We also have $ad = da$ by commutativity. But then $pp = padp = pdap = da$, which is clearly an erroneous conclusion.

Our solution to this paradox is to use w instead of d. The program w can be regarded as an assignment of an undefined value to i. As such, it is a transformation of the local state, much like an ordinary assignment. Since w is a program and not a test, it is not required by the axioms of KAT to commute with tests.

References

1. Ernie Cohen. Hypotheses in Kleene algebra. Available as ftp://ftp.bellcore.com/pub/ernie/research/homepage.html, April 1994.
2. Ernie Cohen. Lazy caching. Available as ftp://ftp.bellcore.com/pub/ernie/research/homepage.html, 1994.
3. Ernie Cohen. Using Kleene algebra to reason about concurrency control. Available as ftp://ftp.bellcore.com/pub/ernie/research/homepage.html, 1994.
4. John Horton Conway. *Regular Algebra and Finite Machines.* Chapman and Hall, London, 1971.
5. Michael J. Fischer and Richard E. Ladner. Propositional dynamic logic of regular programs. *J. Comput. Syst. Sci.*, 18(2):194–211, 1979.
6. Stephen C. Kleene. Representation of events in nerve nets and finite automata. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, pages 3–41. Princeton University Press, Princeton, N.J., 1956.
7. Dexter Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Infor. and Comput.*, 110(2):366–390, May 1994.
8. Dexter Kozen. Kleene algebra with tests. *Transactions on Programming Languages and Systems*, 19(3):427–443, May 1997.
9. Dexter Kozen. Efficient code certification. Technical Report 98-1661, Computer Science Department, Cornell University, January 1998.
10. Dexter Kozen. On Hoare logic and Kleene algebra with tests. *Trans. Computational Logic*, 1(1), July 2000. To appear.
11. Dexter Kozen and Frederick Smith. Kleene algebra with tests: Completeness and decidability. In D. van Dalen and M. Bezem, editors, *Proc. 10th Int. Workshop Computer Science Logic (CSL'96)*, volume 1258 of *Lecture Notes in Computer Science*, pages 244–259, Utrecht, The Netherlands, September 1996. Springer-Verlag.
12. Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *25th ACM SIGPLAN/SIGSIGACT Symposium on Principles of Programming Languages*, pages 85–97, San Diego California, USA, January 1998.
13. George C. Necula. Proof-carrying code. In *Proc. 24th Symp. Principles of Programming Languages*, pages 106–119. ACM SIGPLAN/SIGACT, January 1997.
14. George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proc. Conf. Programming Language Design and Implementation*, pages 333–344. ACM SIGPLAN, 1998.