Annotation inference for modular checkers

Cormac Flanagan, Rajeev Joshi, and K. Rustan M. Leino

Compaq Systems Research Center,
130 Lytton Ave., Palo Alto, CA 94301, U.S.A.

This paper presents a general approach to annotation inference for a given static program checker. The approach reuses the checker as a subroutine. The approach has been used to implement annotation inference systems for two static program checkers, ESC/Java and `rccjava`. The paper describes the approach formally and shows how it applies to ESC.

0. INTRODUCTION

Static program checkers find software defects. Many static checkers rely on the programmer to supply annotations describing program properties such as invariants and module specifications. The annotations permit the checker to find defects using a local (modular) analysis, because the annotations provide specifications of module interfaces. During the analysis, the checker verifies that the supplied annotations are consistent with the program. The presence of the annotations guides the checking process, thus making the checking problem conceptually and computationally simpler.

For example, conventional type checkers follow this modular approach and rely on type annotations to guide the type checking process. Similarly, static race condition checkers, like `rccjava` [7], rely on annotations describing the locking discipline. Another kind of modular checker is an extended static checker like ESC/Modula-3 [3] or ESC/Java [6,8], whose annotations include preconditions, postconditions, and object invariants.

A limitation of the modular checking approach is the burden on the programmer to supply annotations. Although programmers have grown accustomed to writing type annotations, they have been reluctant to provide additional annotations. In our experience, this reluctance has been the major obstacle to the adoption of modular checkers like ESC/Java and `rccjava`. This annotation burden appears particularly pronounced when one is faced with the task of checking an existing (unannotated) program.

In this paper, we present a general approach to building an annotation inference system, an *annotation assistant*, for a given checker. Our approach reuses the checker as a subroutine. We have implemented annotation assistants for both ESC/Java and `rccjava` and have found the inferred annotations to be useful.

In more detail, our annotation assistants have the following form:

> generate candidate annotation set;
> **repeat**
>   invoke checker to refute annotations;
>   remove the refuted annotations
> **until** quiescence

The *candidate annotation set* is a finite set generated from the program text and checker-specific heuristics about what annotations are possible and/or likely to apply to the program. We will not describe the nature of the candidate set or its generation any further in this paper. Instead, we focus on the loop of the pseudo-code above.

Like any invocation of the checker, the invocation inside the loop produces warnings about pieces of the program that violate some of the given annotations. The annotation assistant interprets such warnings as identifying incorrect annotation guesses in the candidate set. In this sense, an invocation of the checker has the effect of refuting some number of candidate annotations.

The net effect of the loop is to remove incorrect candidate annotations. Thus, the set of annotations remaining upon termination is a correct subset of the candidate set.

Some questions immediately arise about the correctness of the annotation assistant algorithm: Does the algorithm terminate with a unique answer? Does it matter in which order the checker is invoked on the various parts of the program? Does the checker need to be applied to all parts of the program on every iteration? What properties of the checker does the annotation assistant rely on? In this paper, we describe the approach formally so that we can answer these questions.

We start in Section 1 by formalizing modular checkers. Section 2 presents the basic annotation assistant algorithm and the subsequent section presents a more efficient algorithm. Section 4 defines a miniature extended static checker and proves that it satisfies the checker properties assumed by the two algorithms. The last couple of sections discuss related work and conclude.

Before going on, we introduce some notational conventions. We write $\mathcal{P}\,X$ to denote the power set of X. Following Dijkstra, we use a left-associative infix "." (binding stronger than any other operator) to denote function application. The expression $\{\, x \mid r.x \;::\; t.x \,\}$ denotes the set of terms of the form $t.x$ for all x satisfying the range expression $r.x$. For $\mathcal{Q}$ denoting $\forall$ or any associative operator that is symmetric on the elements of $\{\, x \mid r.x \;::\; t.x \,\}$ (for example, $\cup$), the expression $\langle\, \mathcal{Q}\, x \mid r.x \;::\; t.x \,\rangle$ denotes the application of $\mathcal{Q}$ to the elements of $\{\, x \mid r.x \;::\; t.x \,\}$. If the range expression is *true*, we may omit the "$\mid$ *true*".

1. MODULAR CHECKERS

A *modular checker* checks a program one part at a time. We refer to the program parts on which the checker operates as *units of checking*, or simply as *units*. For some checkers, a unit of checking may be a routine such as a procedure, method, or constructor. For other checkers, a unit may be a larger construct such as a module, package, or class. We let *Unit* denote the set of possible units of checking. We are not concerned with the internal structure of these units: we simply assume that a program is a finite set of units and that we have a checker that can check these units.

While checking each unit, the checker relies on annotations specifying properties of the other units in the program. We use *Ann* to denote the set of possible annotations.

During the checking process, the checker may produce two kinds of warnings. The first kind of warning concerns potential run-time errors, such as dereferencing the null pointer. Although these warnings are useful for static debugging purposes, they do not aid us in the construction of an annotation assistant, and we will not discuss them further.

The second kind of warning concerns refuted annotations. During the checking process, the checker may discover that the given unit is not consistent with some given annotation (for example, the unit may be a procedure that fails to ensure one of its postconditions). In this case, we say that the checker *refutes* the annotation. We formalize the checker as a function C that takes a unit f and a set of annotations A and returns the set of annotations in A that are not refuted:

$$C\colon \mathit{Unit} \times \mathcal{P}\,\mathit{Ann} \to \mathcal{P}\,\mathit{Ann}$$

An annotation assistant relies on two properties that we assume about the underlying checker. The first is that the set of annotations returned by the checker is a subset of those to which the checker is applied, that is, $C.f$ is *contracting*:

$$\langle \forall\, f, A \mid f \in \mathit{Unit} \wedge A \subseteq \mathit{Ann} \;::\; C.f.A \subseteq A \,\rangle \tag{Ch0}$$

The second property is that $C.f$ is *monotonic*:

$$\langle \forall\, f, A, B \mid f \in \mathit{Unit} \wedge A \subseteq B \subseteq \mathit{Ann} \;::\; C.f.A \subseteq C.f.B \,\rangle \tag{Ch1}$$

Intuitively, if a checker invocation does not refute a particular annotation, then passing additional annotations to the checker does not cause that annotation to be refuted either.

For convenience, we overload ("lift") C to also apply to sets of units: for any subset F of *Unit*,

$$C.F.A = \langle \cap\, f \mid f \in F \;::\; C.f.A \,\rangle \cap A$$

For the lifted checker, $C.F$ is also contracting and monotonic. Furthermore, for any unit f, set of units F, and set of annotations A, we have

$$f \in F \Rightarrow C.F.A \subseteq C.f.A \qquad (0)$$
$$f \in F \wedge C.F.A = A \Rightarrow C.f.A = A \qquad (1)$$

We say that an annotation set A is *valid* for a program P if $C.P.A = A$, that is, if C does not refute any of the annotations in A. It follows from properties Ch0 and Ch1 that validity is closed under union. Hence, for any program P and annotation set A, there is a unique greatest subset of A that is valid for P.

2. ANNOTATION ASSISTANTS

An *annotation assistant* is a program that, for a given (finite) candidate annotation set G and a program P, computes the greatest subset of G that is valid for P. Formally, an annotation assistant computes a set B such that

$$B \subseteq G \qquad \text{(AA0)}$$
$$C.P.B = B \qquad \text{(AA1)}$$
$$\langle \forall X \mid X \subseteq G \wedge C.P.X = X :: X \subseteq B \rangle \qquad \text{(AA2)}$$

The following program implements an annotation assistant.

$B := G$;
while $C.P.B \neq B$ **do**
  choose X **such that** $C.P.B \subseteq X \subsetneq B$;
  $B := X$
end

The body of this loop picks any set X that satisfies the given constraint and then sets B to X. The loop guard and Ch0 guarantee that such an X exists.

The program satisfies the specification of an annotation assistant: It is not hard to prove, using Ch1, that AA0 $\wedge$ AA2 is a loop invariant. The negation of the loop guard is AA1. Termination follows from variant function $|B|$, which is strictly decreased by the loop body.

Note that this program can remove from B any annotation that $C.P.B$ refutes; it need not contract B to $C.P.B$ itself. Thus refuted annotations can be removed from B in any order.

3. A MORE EFFICIENT ALGORITHM

Although the algorithm described in the preceding section is simple, it is also inefficient, since it involves computing C on all units of the program in each iteration. In practice, this is usually unnecessary, since units are often "independent" of certain annotations (*e.g.*, if the units do not read or write the variables mentioned in those annotations). In this section, we explore this idea further in order to develop a more efficient algorithm for computing the greatest valid subset.

A relation *indep* on $\mathit{Unit} \times \mathcal{P}\,\mathit{Ann}$ is an *independence relation* for a checker C if it satisfies the following properties, for all units f and annotation sets B and K,

$$\mathit{indep}.f.B \Rightarrow C.f.(B \cup K) \subseteq B \cup C.f.K \qquad \text{(Ind0)}$$
$$\mathit{indep}.f.\emptyset \qquad \text{(Ind1)}$$

Informally, $\mathit{indep}.f.B$ denotes that unit f is independent of the annotations in B.

Given an independence relation, we develop a more efficient algorithm that recomputes $C.f.A$ for a unit f only upon removal of an annotation on which f depends. This is achieved by using a set W (called the "work set") to record those units that remain to be checked. In particular, we maintain the loop invariant AA0 $\wedge$ AA2 $\wedge$ (2) $\wedge$ (3) where

$$W \subseteq P \qquad (2)$$
$$C.(P \setminus W).B = B \qquad (3)$$

Informally speaking, (3) states that the annotation set B is valid for the set of units $P \setminus W$. Loop invariants (2) and (3) are initially established by setting the work set to P, and whenever a set R of annotations is removed from B, the work set is enlarged to include all the units that are not independent of R. The loop terminates when the work set is empty. The full algorithm is given in Figure 0.

We prove that this program satisfies the specification of an annotation assistant. First, note that the negation of the guard is $W = \emptyset$, which with (3) implies AA1. The proofs that AA0 and AA2 are loop invariants are similar to those of the algorithm in the previous section. It is simple to prove that (2) is a loop invariant and that

$$\begin{array}{l}
B := G \,;\, W := P \,; \\
\textbf{while } W \neq \emptyset \textbf{ do} \\
\quad \textbf{choose } F \textbf{ such that } \emptyset \subsetneq F \subseteq W \,; \\
\quad X := C.F.B \,; \\
\quad R := B \setminus X \,; \\
\quad Y := \{\, f \mid f \in P \wedge \neg \mathit{indep}.f.R \;::\; f \,\} \,; \\
\quad B := X \,; \\
\quad W := (W \setminus F) \cup Y \\
\textbf{end}
\end{array}$$

Figure 0. The more efficient algorithm.

(3) is established initially. Proving that (3) is maintained by the body comes down to proving that

$$C.(P \setminus ((W \setminus F) \cup Y)).X = X$$

holds just before the assignment to B. By Ch0 and the definition of the lifted checker, it is sufficient to prove that

$$\langle \forall f \mid f \in P \setminus ((W \setminus F) \cup Y) \;::\; X \subseteq C.f.X \,\rangle$$

With F, X, R, and Y defined as suggested by the program text, we calculate,

$$\begin{array}{ll}
 & f \in P \setminus ((W \setminus F) \cup Y) \\
= & \{ \text{ set calculus } \} \\
 & f \in P \wedge f \notin W \setminus F \wedge f \notin Y \\
= & \{ \text{ definition of } Y \ \} \\
 & f \in P \wedge f \notin W \setminus F \wedge \mathit{indep}.f.R \\
\Rightarrow & \{ \text{ Ind0 with } B, K := R, X \ \} \\
 & f \in P \wedge f \notin W \setminus F \wedge \\
 & C.f.(R \cup X) \subseteq R \cup C.f.X \\
= & \{ \text{ set calculus and } F \subseteq P \text{ (by (2)), and} \\
 & \quad\; R = B \setminus X \text{ and } R \cup X = B \text{ (by Ch0) } \} \\
 & (f \in P \setminus W \vee f \in F) \wedge \\
 & C.f.B \subseteq (B \setminus X) \cup C.f.X \\
\Rightarrow & \{ \text{ (3), (1), and (0) } \} \\
 & (C.f.B = B \vee C.F.B \subseteq C.f.B) \wedge \\
 & C.f.B \subseteq (B \setminus X) \cup C.f.X \\
\Rightarrow & \{ \text{ lifted Ch0 } \} \\
 & C.F.B \subseteq C.f.B \wedge C.f.B \subseteq (B \setminus X) \cup C.f.X \\
\Rightarrow & \{ \ X = C.F.B \ \} \\
 & X \subseteq (B \setminus X) \cup C.f.X \\
= & \{ \text{ set calculus } \} \\
 & X \subseteq C.f.X
\end{array}$$

source statement S	$ts.X.Y.S$
$v := e$	$v := e$
assert e	**assert** $()\!: e$
assume e	**assume** $()\!: e$
$S0 \,;\, S1$	$ts.X.Y.S0 \,;\, ts.X.Y.S1$
$S0 \;\Box\; S1$	$ts.X.Y.S0 \;\Box\; ts.X.Y.S1$
var w **in** S **end**	**var** w **in** $ts.X.Y.S$ **end**
call m	

$$\begin{array}{l}
\langle \;;\; p \mid (\textbf{pre}, m, p) \in Y \;:: \\
\qquad \textbf{assert } (\textbf{pre}, m, p)\!: p \,\rangle \,; \\
\textbf{var } \grave{Var}, \acute{Var} \textbf{ in} \\
\quad \langle \;;\; v \mid v \in \mathit{Var} \;::\; \grave{v} := v \,\rangle \,; \\
\quad \langle \;;\; q \mid (\textbf{post}, m, q) \in X \;:: \\
\qquad\quad \textbf{assume } (\textbf{post}, m, q)\!: q \,\rangle \,; \\
\quad \langle \;;\; v \mid v \in \mathit{Var} \;::\; v := \acute{v} \,\rangle \\
\textbf{end}
\end{array}$$

Figure 1. The source language (left column) and its translation ts into intermediate-language statements (right column).

Termination of the more efficient program follows from the lexicographically ordered variant function $(|B|, |W|)$, which is strictly decreased by the loop body, as can be proved using Ch0 and Ind1.

4. APPLICATION: EXTENDED STATIC CHECKING

In this section, we formally define a miniature extended static checker, ESC. We show that it satisfies the two properties Ch0 and Ch1, which are assumed of the checker by the basic algorithm in Section 2. We then define independence for ESC annotations, and show that the definition satisfies the properties Ind0 and Ind1, which are used by the more efficient algorithm in Section 3.

For ESC, a unit of checking is a procedure, which has the form

proc m **is** S

and declares that m is the name of a procedure with body S. A procedure body is a source statement in the simple untyped language shown in the left-hand column of Figure 1. The language is a variation of the guarded-command language introduced by Dijkstra [4], with some more recent additions (see, *e.g.*, [11,0]). It includes assignments, assert and assume statements, sequential composition, demonic choice, local variable introduction, and procedure calls. For simplicity, loops are not included, but recursion is allowed.

An execution of a source statement has three possible outcomes: it may terminate (in some state), it may recurse forever, or it may "go wrong". The execution goes wrong if it reaches a statement **assert** e in a program state where the predicate e evaluates to *false*. If e evaluates to *true*, the assert statement terminates without changing the state.

The language includes demonic (blind) nondeterminism, in two ways. First, the execution of $S \;\Box\; T$ executes either S or T, but the choice between the two is made arbitrarily. Second, the execution of **var** w **in** S **end** executes S having first introduced local variables w with arbitrary initial values. The nondeterminism in a program statement can be tamed by assume statements: every "arbitrary" choice is made in such a way that the predicate e is *true* whenever the execution reaches a statement **assume** e. Stated differently, the semantics of a statement considers only those executions in which the predicates of assume statements evaluate to *true*. The assume statement does not change the program state.

For example, the statement

$$(\textbf{assume}\ 0 \le x \ ;\ S) \quad \Box \quad (\textbf{assume}\ x < 0 \ ;\ T)$$

is the familiar deterministic statement

$$\textbf{if}\ 0 \le x\ \textbf{then}\ S\ \textbf{else}\ T\ \textbf{end}$$

In addition to procedure declarations, programs accepted by ESC can have variable declarations. We let *Var* denote the set of variables declared in the program.

The annotations accepted by ESC are pre- and postconditions of the forms:

$(\textbf{pre}, m, p)$
$(\textbf{post}, m, q)$

where **pre** and **post** are keywords, m is a procedure name, p is a predicate over *Var*, and q is a predicate over $\grave{Var}$ and $\acute{Var}$, denoting the

intermediate-language statement S	$wp_a.S.R$
$v := e$	$R(v := e)$
assert $b\colon e$	$\begin{cases} e \wedge R & \text{if } a = b \\ e \Rightarrow R & \text{if } a \neq b \wedge b = () \\ R & \text{if } a \neq b \wedge b \neq () \end{cases}$
assume $b\colon e$	$e \Rightarrow R$
$S0 \ ; S1$	$wp_a.S0.(wp_a.S1.R)$
$S0 \;\Box\; S1$	$wp_a.S0.R \wedge wp_a.S1.R$
var w **in** S **end**	$\langle \forall\, w :: \ wp_a.S.R \,\rangle$ provided no variable in w occurs free in R

Figure 2. The intermediate language (left column) and its weakest-precondition semantics with respect to a label a and post-state predicate R (right column).

values of the variables in the pre- and post-states, respectively. For instance, a postcondition for a procedure *dec* to decrement x may be $(\textbf{post}, dec, \acute{x} = \grave{x} - 1)$. The set *Ann* consists of all annotations of these two forms.

We define the checker ESC by a staged translation, similar to the one used by ESC/Java [8]: we first translate the source statements into an intermediate language (see the left-hand column of Figure 2), and then use weakest preconditions to translate the intermediate-language statements into verification conditions. A verification condition is a predicate whose universal truth is tested by ESC (whose implementation would in practice make use of an automatic theorem prover).

The intermediate language has no procedure call statements. Instead, the translation replaces each call by its meaning according to the called procedure's pre- and postcondition annotations. The assert and assume statements of the intermediate language bear labels, so as to keep track of whether the statement originated in the source or was generated on behalf of some annotation, and if so, which one. The labels of assert statements are used by the checker to identify which annotations to refute.

The first of the two translation stages rewrites source statements by taking annotations into account. In particular, we write $tp.f.X.Y$ to denote

$$\begin{array}{l} tp.(\textbf{proc}\ m\ \textbf{is}\ S).X.Y\ = \\ \quad \langle\ ;\ p\ \mathbf{|}\ (\textbf{pre}, m, p) \in X\ :: \\ \qquad\qquad\qquad \textbf{assume}\ (\textbf{pre}, m, p){:}\, p\ \rangle\ ; \\ \quad \langle\ ;\ v\ \mathbf{|}\ v \in \mathit{Var}\ ::\ \grave{v} := v\ \rangle\ ; \\ \quad ts.X.Y.S\ ; \\ \quad \langle\ ;\ v\ \mathbf{|}\ v \in \mathit{Var}\ ::\ \acute{v} := v\ \rangle\ ; \\ \quad \langle\ ;\ q\ \mathbf{|}\ (\textbf{post}, m, q) \in Y\ :: \\ \qquad\qquad\qquad \textbf{assert}\ (\textbf{post}, m, q){:}\, q\ \rangle \end{array}$$

Figure 3. The translation *tp* of a procedure into the intermediate language.

the procedure f rewritten according to the sets of annotations X (used in assume contexts) and Y (used in assert contexts). Stated differently, the annotations in X may be assumed in the target statement whereas the annotations in Y may give rise to checks in the target statement. Using two annotation sets as arguments to *tp* instead of one is convenient in proofs, as we shall see later. The definition of *tp* ("translate procedure") and its auxiliary function *ts* ("translate statement") are shown in Figures 3 and 1, respectively.

The second stage of the translation of procedures into verification conditions uses a function wp_a, which determines the semantics of intermediate-language statements with respect to a given label a. For any intermediate-language statement S and predicate R on the post-state of S, $wp_a.S.R$ characterizes the pre-states from which executions of S do not go wrong because of a and terminate only in states satisfying R. By defining the semantics with respect to a given annotation, we are able to characterize when a procedure might violate the annotation. (Without this feature, the checker would only be able to detect that the procedure is inconsistent with *some* annotation, but could not report which.) The definition of wp_a is given in Figure 2.

Under wp_a, an assert statement originating in the source program (with the empty label "()") is given the conventional "assert-stop" semantics: if $a = ()$, then the asserted condition is checked in the usual fashion; if $a \neq ()$, then the asserted condition is assumed to hold, because no execution gets past the statement unless the condition does hold. An assert statement with a nonempty label b denoting a guessed annotation is given "assert-continue" semantics: if $a = b$, then the asserted condition is again checked as usual; if $a \neq b$, then the assert statement is ignored, because the annotation should not be relied on until the annotation assistant reaches quiescence without having refuted it. Operationally, if the predicate evaluates to *false*, then an assert-stop check results in abrupt program termination, whereas an assert-continue check may cause some diagnostic message to be logged but does not otherwise affect program execution.

Now we can finally define ESC. For a procedure f and a set of annotations A, ESC returns the subset of A that, assuming all annotations in A, does not cause f to go wrong:

$$\begin{array}{l} ESC.f.A\ = \\ \{\, a\ \mathbf{|}\ a \in A \wedge [wp_a.(tp.f.A.A).\mathit{true}]\ ::\ a\ \} \end{array}$$

The square brackets around the wp_a predicate are "everywhere brackets" [5] and universally quantify over all program variables. Thus, the formula $[p]$ denotes that predicate p is universally *true*.

4.0. ESC is a checker

We now prove that ESC is a checker. The fact that ESC satisfies Ch0 follows directly from the definition of ESC. To prove Ch1, it is useful to introduce *program refinement* [0] for intermediate-language statements.

A statement S is *refined* by a statement T when T has the same behavior as S, except possibly that T goes wrong under fewer circumstances and exhibits less nondeterminism (for example, it may contain more assume statements). Formally, we define refinement with respect to a label a: for any intermediate-language statements S and T,

$$S \sqsubseteq_a T\ \ \equiv\ \ \langle \forall R\ ::\ [wp_a.S.R \Rightarrow wp_a.T.R]\ \rangle$$

where R ranges over predicates on the post-states of S and T.

The function *tp* enjoys the following refinement properties, for any label a, procedure f, and annotation sets X, Y, and Z:

$$tp.f.X.Z \sqsubseteq_a tp.f.(X \cup Y).Z \tag{4}$$
$$tp.f.X.(Y \cup Z) \sqsubseteq_a tp.f.X.Y \tag{5}$$
$$a \notin Z \setminus Y\ \Rightarrow\ tp.f.X.Y \sqsubseteq_a tp.f.X.(Y \cup Z) \tag{6}$$

These properties follow from the definitions of tp and ts.

To prove that ESC satisfies the monotonicity property Ch1, we show

$$ESC.f.A \subseteq ESC.f.(A \cup B)$$

for any procedure f and annotation sets A and B. We calculate,

$$
\begin{array}{ll}
& ESC.f.A \subseteq ESC.f.(A \cup B) \\
= & \{ \text{ definition of ESC } \} \\
& \{ a \mid a \in A \wedge [wp_a.(tp.f.A.A).true] :: a \} \subseteq \\
& \{ a \mid a \in A \cup B \wedge \\
& \qquad [wp_a.(tp.f.(A \cup B).(A \cup B)).true] :: a \} \\
= & \{ \text{ set calculus } \} \\
& \langle \forall a \mid a \in A :: [wp_a.(tp.f.A.A).true] \Rightarrow \\
& \qquad [wp_a.(tp.f.(A \cup B).(A \cup B)).true] \rangle \\
\Leftarrow & \{ \text{ predicate calculus } \} \\
& \langle \forall a \mid a \in A :: \langle \forall R :: [wp_a.(tp.f.A.A).R \Rightarrow \\
& \qquad wp_a.(tp.f.(A \cup B).(A \cup B)).R] \rangle\rangle \\
= & \{ \text{ definition of } \sqsubseteq_a \ \} \\
& \langle \forall a \mid a \in A :: \\
& \qquad tp.f.A.A \sqsubseteq_a tp.f.(A \cup B).(A \cup B) \rangle \\
\Leftarrow & \{ \text{ (4) with } X, Y, Z := A, B, A \ \} \\
& \langle \forall a \mid a \in A :: \\
& \qquad tp.f.(A \cup B).A \sqsubseteq_a tp.f.(A \cup B).(A \cup B) \rangle \\
\Leftarrow & \{ \text{ (6) with } X, Y, Z := (A \cup B), A, B \ \} \\
& true
\end{array}
$$

We have thus shown that ESC is a checker.

4.1. Independence in ESC

We now define an independence relation $indep$ for ESC, which permits us to use the more efficient algorithm of Section 3.

For any intermediate-language statement S, we define $assumes.S$ to be the set of annotations attached to assume statements in S, see Figure 4. Note that

$$assumes.(tp.f.X.Y) \text{ is independent of } Y \tag{7}$$

and

$$\text{as a function of } X,\ assumes.(tp.f.X.Y) \text{ distributes over arbitrary union} \tag{8}$$

Finally, for any procedure f and annotation sets X, Y, and Z,

$$assumes.(tp.f.X.Z) = assumes.(tp.f.Y.Z) \equiv tp.f.X.Z = tp.f.Y.Z \tag{9}$$

intermediate-language statement S	$assumes.S$
$v := e$	$\emptyset$
assert $a{:}\,e$	$\emptyset$
assume $a{:}\,e$	$\begin{cases} \emptyset & \text{if } a = () \\ \{a\} & \text{if } a \in Ann \end{cases}$
$S0 \,;\, S1$	$assumes.S0 \cup assumes.S1$
$S0 \,\square\, S1$	$assumes.S0 \cup assumes.S1$
var w **in** S **end**	$assumes.S$

Figure 4. The definition of $assumes$ for intermediate-language statements.

We define independence for ESC as follows, for any f and B:

$$indep.f.B \ \equiv\ assumes.(tp.f.B.\emptyset) = \emptyset \tag{10}$$

To prove that $indep$ is an independence relation, we next prove that it satisfies Ind0 and Ind1.

For any f and B such that $indep.f.B$, and for any K, we calculate,

$$
\begin{array}{ll}
& ESC.f.(B \cup K) \\
= & \{ \text{ definition of ESC } \} \\
& \{ a \mid a \in B \cup K \wedge \\
& \qquad [wp_a.(tp.f.(B \cup K).(B \cup K)).true] :: a \} \\
= & \{ \text{ split range } \} \\
& \{ a \mid a \in B \wedge \\
& \qquad [wp_a.(tp.f.(B \cup K).(B \cup K)).true] :: a \} \\
& \cup \{ a \mid a \in K \wedge \\
& \qquad [wp_a.(tp.f.(B \cup K).(B \cup K)).true] :: a \} \\
\subseteq & \{ \text{ set calculus } \} \\
& B \cup \{ a \mid a \in K \wedge \\
& \qquad [wp_a.(tp.f.(B \cup K).(B \cup K)).true] :: a \} \\
\subseteq & \{ \text{ (5) with } X, Y, Z := B \cup K, K, B \text{ and} \\
& \quad \text{set calculus } \} \\
& B \cup \{ a \mid a \in K \wedge \\
& \qquad [wp_a.(tp.f.(B \cup K).K).true] :: a \} \\
= & \{ \text{ by calculation below and (9):} \\
& \quad tp.f.(B \cup K).K = tp.f.K.K \ \} \\
& B \cup \{ a \mid a \in K \wedge [wp_a.(tp.f.K.K).true] :: a \} \\
= & \{ \text{ definition of ESC } \} \\
& B \cup ESC.f.K
\end{array}
$$

where we have used

$$
\begin{array}{ll}
& assumes.(tp.f.(B \cup K).K) \\
= & \{ \text{ (8) } \}
\end{array}
$$

$$
\begin{array}{ll}
 & assumes.(tp.f.B.K) \cup assumes.(tp.f.K.K) \\
= & \{\ (7)\ \} \\
 & assumes.(tp.f.B.\emptyset) \cup assumes.(tp.f.K.K) \\
= & \{\ indep.f.B \text{ and } (10)\ \} \\
 & assumes.(tp.f.K.K)
\end{array}
$$

thereby establishing Ind0. Property Ind1 follows directly from (10) and (8) with the empty union.

5. RELATED WORK

Abstract interpretation [2] is a standard framework for developing and describing program analyses. We can view an annotation assistant as an abstract interpretation, where the abstract state space is the power set lattice $\mathcal{P}G$ and the checker is used to compute the abstract transition relation.

As usual, the choice of the abstract state space controls the conservative approximations performed by the analysis. In our approach, it is easy to tune these approximations by choosing the set of candidate annotations appropriately, provided that this set remains finite and that the annotations are understood by the checker.

An interesting aspect of our approach is that the checker can use arbitrary techniques (for example, weakest preconditions in the case of ESC) for performing procedure-local analysis. If these local analysis techniques allow the checker to reason about sets of intermediate states that cannot be precisely characterized using the abstract state space $\mathcal{P}G$, then an annotation assistant may yield more precise results than a conventional abstract interpretation that exclusively uses these abstract states to represent sets of concrete states.

The issue of annotation-based program checkers and associated annotation inference algorithms commonly arises in the study of type systems. In contrast to annotation assistants, most type inference algorithms [10,1,9] do not reuse the type checker. In many cases, this reuse may not be possible, for example, because the type checker may not allow multiple type annotation guesses for a given variable declaration. However, for a type checker that does satisfy the properties Ch0 and Ch1 (such as `rccjava` [7]), the approach outlined in this paper provides a useful method for prototyping a simple, though possibly inefficient, type inference algorithm.

6. CONCLUSION

We have presented an approach for building an annotation assistant for a given program checker. The annotation assistant first determines (guesses) a candidate set of annotations and then iteratively calls the underlying checker to distill the candidate set into its greatest valid subset. In this paper, we described the iterative part of this approach formally. In our formalization, a checker is simply a function that satisfies the two properties Ch0 and Ch1. We showed that these two properties imply that the basic annotation assistant algorithm converges on a unique fixpoint, regardless of the order in which annotations are refuted and removed. We also formalized a notion of independence that leads to a more efficient algorithm. As an example, we formalized a miniature extended static checker and showed that it satisfies the relevant properties.

In this paper, we have not addressed the generation of the candidate set. The candidate set would be derived from the program text and from heuristics specific to a given checker and annotation language. We do permit the candidate set to contain contradictory annotations. For example, a procedure may have the contradictory preconditions $(\mathbf{pre}, m, x < y)$ and $(\mathbf{pre}, m, y < x)$. At least one of these preconditions will be refuted when the annotation assistant checks a call site of this procedure.

Program entry points (such as `main` in Java or C) do not have call sites that are visible to the annotation assistant. Therefore we require that the candidate preconditions for program entry points be correct, *i.e.*, they must hold in the initial state of the program. Similarly, any candidate invariants on global variables must hold in the initial state. From this candidate set, the annotation assistant then infers a subset that holds for all reachable states.

In practice, an extended static checker is either unsound or incomplete or both. We can still develop an annotation assistant for such a checker. An incomplete checker may result in additional candidate annotations being refuted. An unsound checker may result in incorrect annotations being inferred. In either case, the inferred annotations reflect the limitations of the checker and may still be useful for static debugging.

We have built two annotation assistants from the approach we presented, for the checkers ESC/Java [6,8] and `rccjava` [7]. From our experience with these, we find the inferred annotations to be useful. Performance has been adequate for `rccjava`, but it remains a problem in the case of ESC/Java. If an annotation assistant is effective and efficient, one would expect users to gravitate toward using the annotation assistant more often than using the underlying checker directly. This means the annotation assistant will be run many times, with small program or manual annotation changes in between. Hence, we are interested in finding an "incremental" algorithm, where a run of the annotation assistant would make use of information gathered in previous runs. We hope that the formalism we presented here will help in reasoning about incremental algorithms and other more efficient algorithms.

Acknowledgements.

The annotation assistant for ESC/Java (called Houdini for ESC) was developed by two of the authors (Flanagan and Leino) and Yuan Yu. The annotation assistant for `rccjava` (called Houdini for `rccjava`) was developed by Flanagan and Stephen Freund. We'd like to thank the attendees of the IFIP WG 2.3 meeting in Longhorsley, England and our colleagues Ernie Cohen and Lyle Ramshaw for feedback that has helped simplify the theory presented here.

REFERENCES

0. Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.
1. Luca Cardelli. Type systems. *The Computer Science and Engineering Handbook*, pages 2208–2236, 1997.
2. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In POPL 4, pages 238–252, January 1977.
3. David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq SRC, December 1998. Available from `research.compaq.com/SRC/publications/`.
4. Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.
5. Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, 1990.
6. Extended Static Checking home page, Compaq Systems Research Center. On the Web at `research.compaq.com/SRC/esc/Esc.html`.
7. Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. In PLDI'00, *ACM SIGPLAN Notices*, 35(5):219–232, May 2000.
8. K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking Java programs via guarded commands. In Bart Jacobs, *et al.*, editors, *Formal Techniques for Java Programs*, Technical Report 251. Fernuniversität Hagen, May 1999. Also available as Technical Note 1999-002, Compaq SRC, from `research.compaq.com/SRC/publications/`.
9. Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.
10. John C. Mitchell. Type systems for programming languages. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, pages 365–458. The MIT Press/Elsevier, 1990.
11. Greg Nelson. A generalization of Dijkstra's calculus. *ACM TOPLAS*, 11(4):517–561, 1989.