Extending Typestate Checking Using Conditional Liveness Analysis

Robert E. Strom and Daniel M. Yellin, *Member, IEEE*

Abstract—**We present a practical extension to typestate checking which is capable of proving programs free of uninitialized variable errors even when these programs contain conditionally initialized variables where the initialization of a variable depends upon the equality of one or more "tag" variables to a constant. The user need not predeclare the relationship between a conditionally initialized variable and its tags, and this relationship may change from one point in the program to another. Our technique generalizes liveness analysis to conditional liveness analysis.**

Like typestate checking, our technique incorporates a dataflow analysis algorithm in which each point in a program is labeled with a lattice point describing statically tracked information, including the initialization of variables. The labeling is then used to check for programming errors such as referencing a variable which may be uninitialized. Our technique incorporates a more expressive lattice, including predicates of the form: "x is initialized if y equals 2." Because the number of tags per variable is small, the added complexity of the analysis is usually small.

The efficiency of our technique is due, to a large extent, to the fact that we use a backwards analysis of the program (instead of the forward analysis used in the original typestate checking algorithm). Our results suggest that backwards analysis—tracking only those properties which need to hold to make the subsequent statements correct—can be more efficient than forward analysis—tracking all properties which are made true by the preceding statements. We conclude with some additional applications of our techniques to program checking.

Index Terms—**Conditionals, dataflow analysis, liveness analysis, program correctness, typestate checking.**

I. Introduction

A. Typestate Checking

MOST modern programming languages include the notion that program variables have a specific *type*. Compilers for these languages perform *type checking*, which ensures that operations on these variables are type correct. The benefits of type checking are well known, and include more errors being caught at compile time and better code being produced by the compiler.

Typestate can be viewed as an extension to the notion of type. It arises from the realization that, at any point in time, the operations that can be performed on a variable depend not only on the type of the variable, but also upon the *state* of the variable. For instance, one can only read or write a variable of type file if one has already performed the open operation on that variable. As another example, one can only reference a variable (of any type) if that variable has already been assigned. *Typestate checking* [12], [11] is a dataflow analysis technique for verifying that the operations performed on variables obey the typestate rules of the language. In this paper, we will focus on the aspect of typestate checking that guarantees that all variables are initialized before being referenced. It is a straightforward exercise to generalize the ideas in this paper to the more general typestate checking problem.

The benefits of typestate checking are similar to the benefits of type checking: more errors being caught at compile time and better code generation. In particular, when typestate checking is embedded in a compiler, the compiler will reject programs unless it can guarantee that all variables are initialized before they are referenced. This eliminates a class of errors which have unpredictable results, which may remain undetected for a long time, and which are hard to isolate when they do occur. Additionally, the compiler is able to insert finalization code automatically, avoiding the need for run-time garbage collection. Finally, if the entire language is checked, it can be made secure [12], allowing untrusted programs to coexist in a single environment.

The Nil and Hermes languages [15], [13] incorporate typestate explicitly into the language definition: the language definition specifies the allowed order of operations for each data type, function signatures are required to be annotated with typestates,[1] and the compilers implement typestate checking.

B. Extending Typestate Checking

In general, it is an undecidable problem to determine if all variables in a program are initialized before being referenced. Therefore, any algorithm to check for this property must perform some sort of approximation. In the Nil and Hermes typestate tracking algorithm, any program which references a conditionally initialized variable will be rejected. To obtain the effect of conditional initialization, the programmer must define a new variant type that holds the conditionally initialized variable. Before referencing this variable, one needs to perform a `reveal` statement, which essentially causes a run-time check to make sure that the variable is indeed initialized. (If it

Manuscript received April 30, 1990; revised April 28, 1992. Recommended by Mark Moriconi.

The authors are with the IBM T.J. Watson Research Center, Yorktown Heights, NY 10598.

IEEE Log Number 9202417.

[1] These annotations indicate the change of typestate that a parameter will occur in the function body (e.g., will become initialized or become uninitialized). Without these annotations, one could not typestate check a module without seeing the code body of the function being called. With these annotations, one can prove a module to be typestate correct independent of the function bodies being invoked.

0098–5589/93$03.00 © 1993 IEEE

Authorized licensed use limited to: IEEE Xplore. Downloaded on November 21, 2008 at 09:50 from IEEE Xplore. Restrictions apply.

is not initialized, an exception handler will be invoked.) While this has the "virtue" of forcing the programmer to provide more explicit documentation, it is also inconvenient, inflexible, and causes a run-time check.

Consider the program in Fig. 1. At the end of statement 2, c and d are conditionally initialized. Using the original typestate algorithm, the typestate at this program point would be $<init(a), init(b), uninit(c), uninit(d)>$; since c and d are only conditionally initialized, their typestate is "coerced" to *uninit*. The typestate algorithm implemented in the Nil and Hermes compilers would report an error at the point where c and d are conditionally printed.

The corresponding legal Hermes code would need to use a variant type. This program is given in Fig. 2.

In this paper, we define a more general typestate tracking algorithm which is able to prove programs to be typestate correct, even when they conditionally initialize and reference variables. Of course, our new algorithm still has some restrictions on the type of conditional initialization that is allowed, but it is quite general in its applicability. Indeed, this new algorith

m would be able to determine that the program of Fig. 1 is typestate correct.

The method given in this paper allows the initialization of a variable v to be contingent upon the value of other variables, called the *tags* of v. This approach is more general than using variants, as 1) a variable may be conditionally initialized even though it has not been declared as a variant, 2) any number of variables can serve as a tag for a single variable, and 3) the significance of a particular tag may be different at different points in the program. We do all this without significantly increasing the complexity of typestate analysis, and without compromising the requirement for avoiding all uninitialized variable errors at compile-time. It also eliminates the extra run-time checks that are associated with **reveal** statements.

As a result, programmers need not declare as much information. It becomes possible to eliminate the variant datatype, together with its relatively clumsy operations, from the Hermes language. It also becomes possible to use typestate analysis in languages like C or Fortran that lack a discriminated variant type, but in which programmers frequently use conditional initialization.[2] Furthermore, with this approach, the compiler can generate finalization earlier in the program than in previous approaches.

Because our algorithm is based upon a generalization of liveness analysis, an additional contribution of this paper is to present a more precise algorithm for liveness analysis.

C. Related Work

Typestate was introduced by Strom and Yemini in [12], [11]. Embedding typestate within a programming language provides a general framework for specifying what states a datatype must be in for operations on that datatype to be legal. These papers also introduced an algorithm for checking the correctness of programs with respect to a typestate framework.

[2]Of course, detecting initialization becomes harder in languages that contain pointers, due to aliasing. Detecting pointer aliasing is an area of current research, but will not be addressed in this paper.

```
a := i_read();          /* statement 1 */
if a = 0                /* statement 2 */
  then
    b := i_read();      /* statement 2a */
    c := i_read();
  else
    b := i_read();      /* statement 2b */
    d := s_read();
  end if;
i_print(b);             /* statement 3 */
if a = 0                /* statement 4 */
  then
    i_print(c);
  else
    s_print(d);
  end if;
```

Fig. 1. Program containing conditional initializations.

```
tagtype:  enumeration('int', 'string');
vartype:  variant of tagtype(
  'int' -> c:  integer {init},
  'string' -> d:  charstring {init});
v:  vartype;
a := i_read();          /* statement 1 */
if a = 0                /* statement 2 */
  then
    b := i_read();      /* statement 2a */
    /* assign the variant v an integer value */
    unite v.c from i_read();
  else
    b := i_read();      /* statement 2b */
    /* assign the variant v a string value */
    unite v.d from s_read();
  end if;
i_print(b);             /* statement 3 */
if (case of v = 'int')          /* statement 4 */
  then
  /* assert that the variant holds an integer value */
  reveal v.c;
    i_print(v.c);
  else
  /* assert that the variant holds a string value */
  reveal v.d;
    s_print(v.d);
  end if;
```

Fig. 2. Program rewritten to use variants.

Other work, such as that found in [4], [10], have also used dataflow analysis techniques for checking program correctness, including detection of references to uninitialized variables. In [3], Eggert introduces an approach for statically detecting the dereferencing of uninitialized pointers. It has features similar to Hermes' variants.

The algorithms given in the papers referenced above all rely on dataflow analysis to check program correctness. Almost all data flow problems can be made more exact by taking conditionals into consideration. For instance, Wegman and Zadeck's paper on conditional constant propagation [16] uses a lattice to track whether a variable has a known constant value or not at a specific program point. They are able to use constant propagation to detect impossible program paths. As applied to checking for initialization, their algorithm would correctly determine programs to be typestate correct when a variable was initialized along possible paths and uninitialized along only dead paths. It would not, however, deal with cases such as the programming in Fig. 1.

In a previous paper [14], we examined various methods for generalizing typestate, while still limiting the complexity of the required analysis. These methods are similar to other approaches in the literature, including relational dataflow anal-

Authorized licensed use limited to: IEEE Xplore. Downloaded on November 21, 2008 at 09:50 from IEEE Xplore. Restrictions apply.

ysis of Jones and Muchnik [7], qualified dataflow analysis of Holley and Rosen [5], and abstract interpretation of Cousot and Cousot [2]. Although we were able to show useful polynomial time generalizations of typestate, these methods are still fairly inefficient.

Part of the reason for this inefficiency was that we used a forward analysis of the program. Not knowing what variables would actually be conditionally referenced later in the program, we had to keep around a lot of information concerning possible tag relationships.[3] In the worst case, the size of the lattice value that labeled each program point was bounded by a (small) polynomial in the number of variables and the range of these variables.

D. Overview

Our previous research led us to consider a backward analysis of the program. This method is a generalization of *liveness* analysis [1]. A variable v is *live* at a program point p iff there exists a program point p' on the path from p to the program exit and either: 1) p' outputs v, or 2) p' uses v to compute a live variable. An equivalent interpretation of "v is live at p" is the assertion "the program is correct only if the variable v is initialized at p." Our approach is to generalize liveness by tracking assertions of the form "the program is correct only if the variable v is initialized at p whenever *pred* holds." The predicate *pred,* which we call a *liveness condition,* is an assertion on some other variable which tags the liveness of variable v. For instance, in Fig. 1, we find that after statement 2, c must be initialized only if $a = 0, d$ must be initialized only if $a \neq 0$, and a and b are unconditionally live.

To make our technique efficient, we impose the restriction that a liveness condition must be a conjunction, but not a disjunction, e.g., we could have $x \in [1 \cdots 3] \wedge y \in [-\infty \cdots 7, 9 \cdots \infty]$ (which means x equals a value in the range $1 \cdots 3$ and $y \neq 8$) serving as a condition for the liveness of variable v, but not $x \in [1 \cdots 3] \vee y \in [-\infty \cdots 7, 9 \cdots \infty]$.

The organization of the rest of this paper is as follows: In Section II, we give a rigorous description of our technique: we define the predicates which we are to track, and give rules for tracking these predicates in a simple imperative language. In Section III, we describe how finalizations can be generated earlier using the algorithm in this paper. We conclude by discussing possible generalizations and new applications of our technique.

II. Conditional Typestate Analysis

Our algorithm for typestate analysis is obtained as a solution to a (backward) *monotone dataflow framework* (see [8], [9] for formal definitions). To do so, we describe: 1) a semi-lattice L with meet operation $\sqcap$, 2) the initial lattice value associated with the exit node of the program flow graph, and 3) the monotonic functions on L associated with each edge of the program flow graph.

[3] Consider the program fragment `if f() then read(x); a := 1; b := 2 endif;` It may be that `x` is being conditionally initialized and `a = 1` is a tag for this fact. Or it may be that `b = 2` is the tag. Or it may be that both are tags. If, later in the program, we have a conditional reference of the form `if a = 1 then print(x) endif;` it becomes clear that only `a` is the tag for `x`. If, instead of scanning the program from beginning to end we instead scan backwards, we will immediately see that `x` is only conditionally referenced when `a=1`. Hence, we need not guess as to what the tag for `a` is.

A. Intervals and Ranges

This section provides definitions we will use in the next section to define our lattice.

An *interval* $w \cdots x$ over the domain of integers denotes all integer values between w and x (including w and x), where $w \leq x$. When $w = -\infty$, there is no lower bound; similarly, when $x = \infty$, there is no upper bound. The interval $y \cdots z$ is said to be *contiguous* to the interval $w \cdots x$ if $y = x + 1$. The interval $y \cdots z$ is said to *overlap* the interval $w \cdots x$ if $w \leq y \leq x \leq z$. We call $w(x)$ the lower (upper) bound of the interval $w \cdots x$.

A *range* is a list of intervals $[lb_1 \cdots ub_1, lb_2 \cdots ub_2, \cdots, lb_k \cdots ub_k]$ over the integers such that $ub_i < lb_{i+1}\, (1 \leq i < k)$.

Given a range r, we write $e \in r$ iff there exists an interval $w \cdots x$ in r and $e \in w \cdots x$. For any ranges r_1 and r_2, we define $r_3 = union(r_1, r_2)$ to be the range such that $e \in union(r_1, r_2)$ iff $e \in r_1 \vee e \in r_2$.

The union of a set of ranges is computed by taking the union of the intervals in each range, merging all contiguous or overlapping ranges, and ordering the resultant intervals appropriately.

Example: The union of $[1 \cdots 3, 5 \cdots 7]$ and $[-\infty \cdots -3, 10 \cdots \infty]$ is $[-\infty \cdots -3, 1 \cdots 3, 5 \cdots 7, 10 \cdots \infty]$. The union of $[1 \cdots 3, 6 \cdots 7]$ and $[2 \cdots 4, 8 \cdots 10]$ is $[1 \cdots 4, 6 \cdots 10]$.

B. The Lattice

The lattice L that we use is a refinement of the lattice traditionally used for liveness analysis. Each lattice point in L is of the form $<v_1 : pred_1, v_2 : pred_2, \cdots, v_m : pred_m>$, where each v_i is a program variable, and each $pred_i$ is a predicate which is either

- *true,*
- *false,* or
- a conjunction of one or more *conditions* of the form $v_j \in range$. The variable v_j appearing in a condition is called a *tag* for v_i.

Notice that *true* can be viewed as a conjunction of zero conditions.

The interpretation of labeling a program point p with a lattice point $<v_1 : pred_1, v_2 : pred_2, \cdots, v_m : pred_m>$ is to assert that the program is correct only if, at point p, for each variable $v_i, pred_i \Rightarrow initialized(v_i)$; i.e., if $pred_i$ is true at p, then v_i must be initialized at p. Notice that the assertion "v_i must be initialized at p" is equivalent to the statement "v_i is live at p." In particular, the special case $pred_i = true$ corresponds to the assertion that v_i is unconditionally live (must be initialized), and the special case $pred_i = false$ corresponds to the assertion that v_i is dead (need not be initialized).

A condition of the form $v_j \in [x \cdots x]$ for some integer x will be abbreviated as $v_j = x$; a condition of the form $v_j \in [-\infty \cdots x-1, x+1 \cdots \infty]$ will be abbreviated as $v_j \neq x$.

The v_i component of a lattice point $l \in L$ is denoted $l \cdot v_i$, and takes the form of a predicate $pred_i$, asserting under what

Authorized licensed use limited to: IEEE Xplore. Downloaded on November 21, 2008 at 09:50 from IEEE Xplore. Restrictions apply.

conditions v_i must be initialized. The set of possible values of $pred_i$ form a semi-lattice L_{v_i}. In order to describe the structure of the semi-lattice L, we first describe the structure of the semi-lattice L_{v_i}.

The top element in L_{v_i} is *false* and the bottom element is *true*. For any pair of predicates $l_1, l_2 \in L_v$, we define $l_1 \leq l_2$ iff $l_2 \Rightarrow l_1$ (l_2 logically implies l_1). The *meet* of any pair of lattice points l_1 and l_2 is $l_1 \vee l_2$, provided that this point is expressible in the required form. Otherwise, the meet is the "best approximation" within the lattice to the predicate $l_1 \vee l_2$.

Examples:

$l_1 =$	*true*	$a = 2$	$a = 2 \wedge b = 3$	$a = 2$
$l_2 =$	*false*	$b = 2$	$a = 2 \wedge c \neq 2$	$a \in [-\infty..1, 4..\infty]$
$l_1 \sqcap l_2 =$	*true*	*true*	$a = 2$	$a \neq 3$

The meet can be computed as follows.

- For any $l, l \sqcap false = l$.
- Otherwise, variable t is a tag in $l_1 \sqcap l_2$ only if t is a tag in both l_1 and l_2.
- If t is a tag in both l_1 and l_2 such that $t \in range_1$ is a condition in l_1 and $t \in range_2$ is a condition in l_2, then $t \in union(range_1, range_2)$ is a condition in $l_1 \sqcap l_2$.
- An empty set of conditions is the same as *true*.

Besides the meet, another operation we will sometimes perform on a lattice point l is to conjoin another predicate *pred* with l. This produces the new lattice point $l \wedge pred$. Notice that if l is false, or if l and *pred* are mutually exclusive, then $l \wedge pred$ is *false*. For instance, if l is $x \in [-\infty \cdots 4, 7 \cdots \infty]$ and *pred* is $x \in [5 \cdots 6]$, then $l \wedge pred = false$.

The semi-lattice L is defined as the cross-product semi-lattice $L = <L_{v_1}, \cdots, L_{v_m}>$, whose meet is simply the componentwise meet of the predicates associated with each variable.

The lattice value associated with (the entry to) a node n of a program flow graph will be denoted l_n. We call l_n the *typestate* at n. The component of the lattice value associated with variable v at node n ($l_n \cdot v$) will be called the *typestate* of v at n. For instance, if $l_n \cdot v = true$, then v is unconditionally live at entry to n.

Analysis begins with an assignment of *false* to the typestate of each variables at the exit node, indicating that each variable is dead at exit. For the program to be correct, the analysis must terminate with a typestate of *false* (dead) assigned to all variables at the entry node. The program is illegal if any variable is live or conditionally live at the entry node. By our interpretation, such a variable would have to be initialized at a point where all variables are known to be uninitialized.

While a primary goal of typestate analysis is to determine whether or not all variables are dead at the entry node or not, an additional consequence of applying typestate analysis is to generate finalization code, as we shall discuss in Section III.

C. Program Flow Graph Functions

To simplify this description of conditional typestate, we apply it to a small language containing only assignment, input, output, conditional, and loop statements.

In a dataflow problem, one associates with each edge $e = <p, s>$ in the program flow graph a monotonic function $f_e : L \rightarrow L$. Since we are doing backwards flow analysis, f_e maps a typestate l_s at the successor node into a typestate l_p at the predecessor node. The function f_e depends upon the operation at p. We define f_e for each operation in our simple language. When defining f_e, we describe the typestate at p for each variable v whose typestate at p differs from its typestate at s. For all other variables v', the typestate of v' is the same at p as at s. It is straightforward to verify that each function f_e is indeed monotonic.

Assignment Statements: The effect of an assignment statement $v_t := \mathtt{f}(v_{s_1}, \cdots, v_{s_k})$ on the typestate of a variable x depends upon whether x is the target variable (v_t), a source variable (v_{s_i}), or some other variable.

First consider the target variable v_t. If v_t is not also a source variable (e.g., $v := v+1$), then the typestate of v_t on entry (that is, $l_p \cdot v_t$) will always be *false* since this assignment fulfills any obligation to initialize v_t. If v_t is also a source variable, then its typestate on entry is given by the rules for source variables described below. In any case, if $l_s \cdot v_t$ is *false* (i.e., dead), then the assignment is superfluous—it can be ignored. If $l_s \cdot v_t$ is *pred*, the assignment need only be executed when *pred* is true. This will be discussed in more depth later in this section.

Next, consider a source variable v_{s_i}. A source variable v_{s_i} is required to be live on entry if either: 1) v_t is live on exit, so v_{s_i}'s value is needed to compute v_t, or 2) v_{s_i} is live on exit, so v_{s_i}'s value is needed in any case. We therefore want the lattice point $l_p \cdot v_{s_i}$ to be the highest semi-lattice point implied by both the predicates for v_{s_i} and v_t. This point is obtained by taking the meet of $l_s \cdot v_{v_t}$ and $l_s \cdot v_{s_i}$.

The following table illustrates the typestate $l_p \cdot v$ for different values of $l_s \cdot v$ for an assignment to x which references z.

$l_s.x =$	*true*	*false*	$a = 1 \wedge b = 2$
$l_s.z =$	$b = 2 \wedge c = 2$	$b = 2 \wedge c = 2$	$b = 2 \wedge c = 2$
$l_p.x =$	*false*	*false*	*false*
$l_p.z =$	*true*	$b = 2 \wedge c = 2$	$b = 2$

Finally, we consider variables that are neither the target nor a source of the assignment statement, whose typestate can nevertheless be affected by the assignment. In particular, suppose that at s, v_t is a tag for some variable v. That is, $l_s \cdot v$ has the form $\cdots v_t \in range_t \wedge \cdots$. Then $l_p \cdot v$ will depend upon the nature of the assignment. We distinguish three cases: 1) assignments of the form $v_t := c$ for some constant c, 2) assignments of the form $v_t := v_s$, and 3) all other assignments.

1) Suppose the statement at p is $v_t := c$. In the case where $c \in range_t, v_t$ can be removed as a tag for v. The predicate indicates that if $v_t \in range_t$ (and perhaps some other conditions hold), v must be initialized. As this assignment makes $v_t \in range_t$ true, this condition can be removed from the predicate. In the case where $c \notin range_t$, then $l_p \cdot v$ becomes *false*. The predicate indicates that v need only be initialized when $v_t \in range_t$, but the assignment makes this predicate false.

The following table shows the typestate of v before and after the assignment $x := 3$ for different typestates $l_s \cdot v$ in

Authorized licensed use limited to: IEEE Xplore. Downloaded on November 21, 2008 at 09:50 from IEEE Xplore. Restrictions apply.

which x is a tag.

$l_s.v =$	$x = 3 \wedge y = 2$	$x \neq 2 \wedge y = 2$	$x = 2 \wedge y = 2$
$l_p.v =$	$y = 2$	$y = 2$	$false$

2) Suppose the statement at p is $v_t := v_s$. Then if v must be initialized only if v_t has some property after the statement, it must be initialized only if v_s has the same property before the statement. So we just substitute v_s for v_t if v_t appears as a tag in the typestate. This may require simplification (i.e., if v_s was already a tag for v).

The following table shows the typestate of v before and after the assignment $x := y$ for different typestates $l_s \cdot v$ in which x is a tag.

$l_s.v =$	$x = 2 \wedge a = 2$	$x \neq 2 \wedge y \neq 3$	$x = 2 \wedge y \neq 3$	$x = 2 \wedge y = 3$
$l_p.v =$	$y = 2 \wedge a = 2$	$y \in [-\infty..1, 4..\infty]$	$y = 2$	$false$

3) If the statement has any other form, then we drop v_t as a tag in $l_p \cdot v$, i.e., we remove the conjunct $v_t \in range_t$ from the typestate of $l_p \cdot v$. Since we cannot determine if the assignment to v_t makes the conjunct true or false, we pessimistically require v to be initialized regardless of the value of v_t.

In the description given here, we actually require that superfluous assignments be removed and conditionally needed assignments be transformed to execute precisely when needed. To see why this is so, consider the statement $z := a + b$, with posttypestate $<a : false, b : false, z : false>$. By the rules given in this section, the typestate upon entry to this statement will also be $<a : false, b : false, z : false>$. If a is never initialized before this statement is executed, the program will be found to be typestate correct, but during execution of the statement, a's value will be undefined. Hence, we actually require that this statement be removed from the generated code. If introducing these transformations is undesirable, it would not be hard to change our computation of the typestate so that these transformations would not be necessary.[4]

Input Statements: Since the statement `input` v assigns to v an arbitrary value, the typestate on entry to input v can be computed in the same way the typestate on entry to $v := expr$ is computed, where $expr$ is an arbitrary expression. The rules of the last section show how this computation is done.

Output Statements: The output statement `print` v requires that its argument be initialized. The typestate of v is made *true* (unconditionally live) on entry *print* to a statement.

Conditionals: A conditional statement has the form `if` $expr$ `then` $statements_1$ `else` $statements_2$ `end if`. The entry node p has two exit edges: the edge $<p, s_1>$ leading to $statements_1$ when $expr = true$, and the edge $<p, s_2>$ leading to $statements_2$ when $expr = false$.

[4]For instance, we could change the rules so that an assignment to a variable z would require each source variable to be initialized upon entry to the assignment statement, regardless of whether or not z is live or dead after the assignment statement.

The typestate at the entry node p of a conditional is obtained by taking the meet of the typestates $f_{<p,s_1>}(l_{s_1})$ and $f_{<p,s_2>}(l_{s_2})$ since a variable is only required to be initialized upon entry to the conditional if it must be initialized on either branch of the conditional.

For each source variable v appearing in $expr$, the functions $f_{<p,s_1>}$ and $f_{<p,s_2>}$ create a typestate *true*. For every other variable, the functions $f_{<p,s_1>}$ and $f_{<p,s_2>}$ are identity functions unless $expr$ has a form consistent with a test of a tag variable, that is, $expr$ is of the form $v = c, v \neq c, v \leq c$, or $v \geq c$ for some constant c. In this case, $f_{<p,s_1>}$ conjoins $v = c, (v \neq c, v \in [-\infty \cdots c]$ or $v \in [c \cdots \infty]$, respectively) to the typestate of all variables except for v; $f_{<p,s_2>}$ conjoins $v \neq c, (v = c, v \in [c + 1 \cdots \infty]$, or $v \in [-\infty \cdots c - 1]$, respectively) to the typestate of all variables except for v. These functions indicate that a variable that is live on one branch of the conditional is live on entry to the conditional only if that branch is taken.

The following table shows $f_{<p,s_1>}(l_{s_1}) \cdot v, f_{<p,s_2>}(l_{s_2}) \cdot v$, and their meet $l_p \cdot v$ for different values of $l_{s_1} \cdot v$ and $l_{s_2} \cdot v$ when the expression of the conditional is `if` $x = 3$.

$l_{s_1}.v =$	$true$	$x = 2 \wedge y = 3$	$a = 2 \wedge b = 3$
$l_{s_2}.v =$	$false$	$x = 2 \wedge z = 3$	$a = 2 \wedge c = 3$
$f_{(p,s_1)}(l_{s_1}).v =$	$x = 3$	$false$	$x = 3 \wedge a = 2 \wedge b = 3$
$f_{(p,s_2)}(l_{s_2}).v =$	$false$	$x = 2 \wedge z = 3$	$x \neq 3 \wedge a = 2 \wedge c = 3$
$l_p.v =$	$x = 3$	$x = 2 \wedge z = 3$	$a = 2$

Of course, in the conditional `if` $x = 3, x$ is itself referenced, so the typestate $l_1 \cdot x$ is made *true*.

Example: We can apply what we have discussed so far to our earlier example, Fig. 1. The typestate on entry to statement 3 is $<a : true, b : true, c : a = 0, d : a \neq 0>$. The typestate at entry to statement 2a is $<a : true, b : false, c : false, d : a \neq 0>$; the typestate at entry to statement 2b is $<a : true, b : false, c : a = 0, d : false>$. The test $a = 0$ conflicts with the $a \neq 0$ in the tag for d; the inverse test $a \neq 0$ conflicts with the $a = 0$ in the tag for c, so the typestate on entry to statement 2 is simply $<a : true, b : false, c : false, d : false>$.

Since statement 1 assigns a, the entry typestate has all variables dead, and the program is legal.

Loops: The loop `while` $expr$ `repeat` $statements$ `end while` can be expanded into a flow graph containing a conditional test; before entering the loop body, a conditional determines whether or not the loop body is reexecuted. Therefore, no new functions f_e need to be introduced. For example, the program flow graph for the program of Fig. 3 is given in Fig. 4.

However, it may occasionally be necessary to analyze the loop more than once in order to reach a solution. The worst case complexity analysis is discussed in Section II-D.

Fig. 3 illustrates a program in which this iteration takes place. The typestate of z entering statement 3 is $<z : x = 3 \wedge b = 2>$. At the bottom of the `while` loop, it is $<z : a = 5 \wedge x = 3 \wedge b = 2>$ during the first iteration of typestate assignment. (This is because the exit from the `while` loop corresponds to a branch of the implicit `if` statement of the form "`else a = 5`.") At the entry to the

Authorized licensed use limited to: IEEE Xplore. Downloaded on November 21, 2008 at 09:50 from IEEE Xplore. Restrictions apply.

```
while (a ≠ 5) repeat                    /* statement 1 */
  ...
  if (a = 3)                            /* statement 2 */
    then x := 3; a := 4;
    else a := 5 end if;
  end while;
if (x = 3)                              /* statement 3 */
  then if (b = 2) then print z end if; end if;
```

Fig. 3. A loop requiring more than one pass.

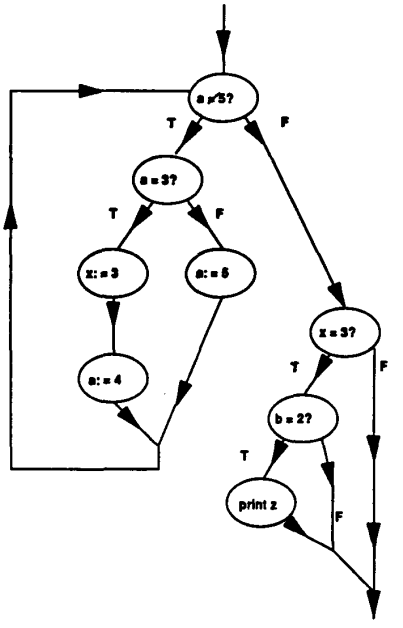


Fig. 4. The program flow graph for the program of Fig. 3.

`then` clause, it is *false;* at the entry to the `else` clause, it is $<z : x = 3 \wedge b = 2>$. At the entry to statement 2, it becomes $<z : a \neq 3 \wedge x = 3 \wedge b = 2>$. Prior to the `while` test, it is $<z : a \neq 3 \wedge a \neq 5 \wedge x = 3 \wedge b = 2>$. Taking the meet with the typestate computed previously after statement 2, one gets $<z : a \neq 3 \wedge x = 3 \wedge b = 2>$. The second time around, the typestate on entry to the `then` clause is lowered from *false* to $<z : b = 2.>$ The typestate of z on entry to statement 2 also becomes $<z : b = 2>$. This is the correct fixed point and is the typestate of z at entry to statement 1.

D. Complexity

To implement our dataflow analysis problem, we use a work-list algorithm [8], [6]. This algorithm will only recompute the typestate at a node (*visit* the node) if the typestate at a successor node has changed values. Since each node of our dataflow graph has a bounded number of successors (two, to be precise), the total number of visits to nodes of the graph is proportional to the number of nodes in the graph times the numbers of typestate changes that can occur at a node.

Since we are dealing with a monotone dataflow analysis framework, a node n changing its typestate is equivalent to its taking on a lower value in the semi-lattice. This means that for some variable v, either a tag for v is dropped from the typestate or the range associated with a tag becomes larger. The latter is equivalent to adding an interval to the range associated with that tag. The number of intervals that can be added to a range of a typestate $l_n \cdot v$ at a node n is bounded by the number of conditionals reachable from n. This is because the upper or lower bound c in any interval added to the range of the typestate must arise from a reachable conditional of the form $q\ op\ c$.

Since the number of tags for a variable v can be as large as the number of variables in the program, the total number of times a typestate $l_n \cdot v$ can change is VC_n, where V is the number of variables in the program and C_n is the number of conditionals reachable from node n. This implies that the number of times a typestate l_n can change is V^2C_n, and the worst case complexity of the program is $O(V^2N^2)$, where N is the number of nodes in the dataflow graph.

In practice, however, we believe that the actual cost of the algorithm in this paper is $O(kVN)$ for some small constant k. This is because, for most programs, the number of variables that are conditionally live is small, and the number of tags for these variables is also small. Since any dataflow algorithm that tracks all variables must do work proportional to $O(VN)$, including the original typestate algorithm, conditional typestate analysis provides improved generality without a large increase in complexity.

For some restricted programs, one can formally show that the complexity is better behaved than the worst case bounds given above. These programs loosely correspond to programs where Hermes-like variants are replaced by tag variables. More formally, consider programs in which one can partition its variables into a set of tags and a set of regular variables. If: 1) for all the conditionals in the program of the form $v = constant, (v \leq constant$ etc.), v is a tag variable, and 2) assignment to tag variables can only reference constants, then one can show that the complexity of the program is bounded by $O(CVN)$, where C is the number of conditionals in the program and V and N are as given above. We now sketch a proof of this fact.

Consider any typestate $l_n \cdot v$ in the program. We argue that only tag variables appear in the typestate $l_n \cdot v$, and that if an interval $x \cdots w$ is added to the range of the tag t in $l_n \cdot v$ causing the typestate to change, then there must exist conditionals reachable from n involving the tag t and the constant x or w (one of x or w may be $-/+\infty$). If this statement is true, then the number of times the typestate $l_n \cdot v$ can change is equal to the number of conditionals reachable from n. This is at most C, the number of conditionals in the program. Hence, the entire typestate l_n can change at most $O(CV)$ times. Therefore, the total complexity is at most (CVN).

To see why the above statement is true, we note that only conditionals and assignments introduce new tags and/or increase the range associated with a tag in a typestate. (Recall that loops are modeled using conditionals.) Since, by 1) above, for all conditionals of the form $t = const\ (t \leq const$, etc.), t is a tag variable, it is straightforward to verify that the conditional will only introduce the tag t with a range involving the constant *const* into the entry typestate $l_n \cdot v$ for any variable v. Hence, we need only concern ourselves with assignment statements.

Authorized licensed use limited to: IEEE Xplore. Downloaded on November 21, 2008 at 09:50 from IEEE Xplore. Restrictions apply.

The only potential problem with an assignment statement is that it may "transfer" the range associated with one tag variable in a typestate to another tag variable, e.g., the typestate $v : t_1 = const$ may become $v : t_2 = const$, even though there is no conditional involving t_2 and the constant *const*. This can occur if the assignment is of the form $v_t := v_s$. In this case, the rules given above call for substituting v_s for v_t in any typestate with a v_t tag. However, since by 2) above, the assignment to any tag variable can only reference constants, this sort of assignment statement will never arise in the restricted programs under consideration.

III. Coercions

One important benefit of typestate analysis is that it enables the compiler to automatically insert storage deallocation (finalization) code. This is especially important when a program's execution can be interrupted in a number of different places due to an exceptional condition.

```
block
  call f(x);
  new a;
  ...
  call f(x);
  ...
  new b;
  ...
  call f(x);
  ...
on exception(f.error);
  ...
end block;
```

In this example, `error` is an exception that may be raised by the function `f`. Suppose that the variables a, b, etc., are large data structures whose storage is dynamically allocated and freed. If an exception occurs, the exception handler may be entered either with a and b both initialized, or with a alone initialized, or with neither a nor b initialized. In Nil/Hermes [15], [13], neither a nor b is allowed to be referenced in the exception handler—both variables are treated as uninitialized.

At the end of the program, we require all initialized variables to be finalized. We could enforce this by run-time garbage collection. However, we can also use typestate analysis to insert static finalization when it can be statically determined that a data structure is dead. Since we are assuming that a and b will be uninitialized in the exception handler, we insert operations to guarantee that a and b are uninitialized on entry to the exception handler, regardless of the path taken. In the above example, a discard (finalization) of a is inserted on the path between the second call of `f` and the handler, and a discard of a and b is inserted on the path between the third call of `f` and the handler. The inserted operations are called *coercions.*

Automatic finalization proved extremely practical when it was implemented in Nil and Hermes. The use of coercions was generalized to other kinds of finalization besides freeing storage (e.g., closing open files). In these languages, dangling references to memory cannot occur nor can programs terminate without either freeing up or passing off resources they own. Coercions give the run-time environment the option of reclaiming memory eagerly rather than performing periodic reclamation (garbage collection). In Nil and Hermes, the determination of where to put coercions was made by the same forward analysis algorithm used to check for typestate correctness.

Backwards dataflow analysis can also be used to generate coercions. In fact, using backwards analysis, we sometimes finalize a variable at an earlier point in the program (thereby freeing storage sooner). In what follows, we give the general rules for how these finalizations could be generated. Sometimes it may be preferable not to finalize data structures earlier (e.g., because the allocated structure can be reused or because extra code may be required that slows the application). This paper does not address the issue of when early finalizations should be used.

Two situations give rise to coercions. The first situation occurs whenever for some edge $e =< p, s >$, with pretypestate l_p and posttypestate l_s, there exists a lower (less "dead" than l_s) typestate l'_s such that $f_e(l'_s) = l_p$. In this case, we insert a coercion operation C after p such that C has a posttypestate of l_s and a pretypestate of l'_s.[5]

$\{ l_p \}$
`p: statement;`
$\{ l'_s \}$
C
$\{ l_s \}$

Here is the simplest example: Suppose there is a statement `print` v with posttypestate $v : false$ and pretypestate $v : true$. That is, the variable v is not needed after it is printed. The function associated with the `print` statement will map a posttypestate of $v : true$ to the same pretypestate of $v : true$. Hence, by the rule given above, we insert a coercion (after the `print` statement) with a pretypestate of $v : true$ and a posttypestate of $v : false$. We call this coercion `discard` v. This guarantees that v will be finalized immediately after the `print` statement.

What if the posttypestate were $v : a = 2$? In that case, the value of v will still be needed if a is 2, but will not be needed if a is not 2, so the coercion is `if` $a \neq 2$ `then discard` v. The tracking of conditional liveness enables us to finalize v early whenever it is correct to do so. Using forward analysis, in which initialization rather than liveness is tracked, v would be finalized at some later point where the path merged with some other path along which v was uninitialized.

The second situation in which coercions must be inserted occurs when two (backwards) paths join and the meet of the two typestates is taken. For example, suppose that on two branches of an `if` statement, the typestates are $v : a = 2 \wedge b = 3$ and $v : b = 3 \wedge c = 4$. Then the meet is $v : b = 3$. The statement *if* $(a \neq 2) \wedge (b = 3)$ `then discard` v will be generated on entry to the left branch, and the statement `if` $(c \neq 4) \wedge (b = 3)$ `then discard` v will be generated on entry to the right branch. (The code for finalizing v when $b = 3$ will occur somewhere above the conditional when the

[5] One way to understand this is as follows: since $f_e(l'_s) = l_p$, the operation does not cause l_p to become uninitialized to l_s, but only to l'_s. Hence, we need to insert a code to perform the missing finalization.

Authorized licensed use limited to: IEEE Xplore. Downloaded on November 21, 2008 at 09:50 from IEEE Xplore. Restrictions apply.

typestate for v goes from $v : b = 3$ to $v : false$.) Once again, this assures that variable v will be finalized at an earlier point than traditional (forward) typestate analysis.

Finalization coercions are often viewed as simply an optimization which enables storage to be freed sooner. However, conditional liveness can be extended to other cases in which finalization has an actual semantic effect. For example, a language could support variables of type *file,* with operations *open, read, write,* and *close.* The same analysis can be used to guarantee that files are closed on program termination by generating *close* statements at appropriate points.

IV. Future Work and Conclusion

A. Other Applications

The framework given in this paper is useful for checking program correctness (that all variables referenced in a program have been defined). It is equally useful for optimizing compilers that make use of liveness information, as the technique given in this paper will provide more precise information than the traditional liveness algorithm.

It is possible to generalize the framework given here to other applications as well. The common property of these applications is that when a statement (e.g., `print` z) is encountered, an expectation is generated for what must precede in order for the statement to be legal (e.g., an assignment to z). Other types of statements could generate other types of expectations which could be encoded as typestates and tracked using a similar conditional analysis.

For example, suppose we wished to weaken the requirement that the type of a variable x remains fixed throughout the program. The variable could be, for example, a *real* in some contexts and a *string* in others. At some places, it could even be either *real* or *string,* depending on the value of variable a. Without testing this tag, it would be illegal to perform operations which assumed one or the other type. The properties *real* and *string* become typestate properties rather than type properties. Values of the lattice will have a form such as $<b = 0 \Rightarrow string(x); b \neq 0 \Rightarrow real(x)>$.

B. Conclusion

Experience with Nil and Hermes has shown typestate checking to be an extremely valuable tool for detecting errors, generating finalizations, and assuring security in a programming language.

To make typestate checking practical for languages like C, stronger analysis techniques must be used to handle conditional initialization. The techniques presented here provide the needed extra strength without undue cost. They also suggest potential useful generalizations to other dataflow analysis problems.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools.* Reading, MA: Addison-Wesley, 1986.

[2] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proc. 4th ACM Symp. Principles of Programming Languages,* ACM, 1977, pp. 238–252.

[3] P. R. Eggert, "Detecting software errors before execution," Ph.D. dissertation, UCLA, 1980.

[4] L. D. Fosdick and L. J. Osterweil, "Dataflow analysis in software reliability," *ACM Computing Surveys,* vol. 8, pp. 305–330, Sept. 1976.

[5] L. H. Holley and B. K. Rosen, "Qualified data flow problems," *IEEE Trans. Software Eng.,* vol. SE-7, pp. 60–78, Jan. 1981.

[6] S. Horwitz, A. Demers, and T. Teitelbaum, "An efficient general iterative algorithm for data flow analysis," *Acta Informatica,* vol. 24, pp. 679–694, 1987.

[7] N. D. Jones and S. S. Muchnik, "Complexity of flow analysis, inductive assertion synthesis and a language due to Dijkstra," in *Program Flow Analysis: Theory and Practice.* Englewood Cliffs, NJ: Prentice-Hall, 1981, pp. 380–393.

[8] G. A. Kildall, "A unified approach to global program optimization," in *Proc. 1st ACM Symp. Principles of Programming Languages,* ACM, Oct. 1973, pp. 194–206.

[9] T. J. Marlowe and B. G. Ryder, "Properties of dataflow frameworks," *Acta Informatica,* vol. 28, pp. 121–163, 1990.

[10] K. Olender and L. Osterweil, "Cecil: A sequencing constraint language for automatic analysis generation," *IEEE Trans. Software Eng.,* vol. 16, pp. 268–280, Mar. 1990.

[11] R. Strom and S. Yemini, "Typestate: A programming language concept for enhancing software reliability," *IEEE Trans. Software Eng.,* vol. SE-12, pp. 157–171, Jan. 1986.

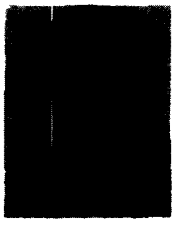
[12] R. E. Strom, "Mechanisms for compile-time enforcement of security," in *Proc. 10th ACM Symp. Principles of Programming Languages,* ACM, Jan. 1983.

[13] R. E. Strom, D. F. Bacon, A. Goldberg, A. Lowry, D. Yellin, and S. A. Yemini, *Hermes: A Language for Distributed Computing.* Englewood Cliffs, NJ: Prentice-Hall, 1991.

[14] R. E. Strom and D. M. Yellin, "Computationally tractable semilattices for global data flow analysis," Tech. Rep. RC 14936, IBM T. J. Watson Res. Cen., Aug. 1989.

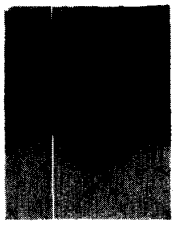
[15] R. E. Strom and S. A. Yemini, "NIL: An integrated language and system for distributed programming," in *Proc. SIGPLAN'83 Symp. Programming Language Issues in Software Syst.,* June 1983.

[16] M. Wegman and F. K. Zadeck, "Constant propagation with conditional branches," *ACM Trans. Programming Languages Syst.,* vol. 13, pp. 181–210, Apr. 1991.



Robert E. Strom completed undergraduate studies at Harvard in 1966 and doctoral studies at Washington University in 1971.

Since 1977 he has been a Research Staff Member at the IBM T.J. Watson Research Center. His interests include programming languages, operating systems, distributed computing, fault-tolerant computing, and end-user environments. He is a member of the Distributed Systems Software Technology Group at IBM, which is developing a high-level language-based platform for heterogeneous distributed computing. He is the principal designer of the Hermes distributed programming language.



Daniel M. Yellin (S'85–M'87) received the Ph.D. degree in computer science from Columbia University in 1987.

Since 1987 he has been a Research Staff Member at the IBM T.J. Watson Research Center in the Distributed Software Technology Group. His interests include programming languages and environments, distributed computing, algorithms, and data structures. He is an Editor of the International Standards Organization (ISO) standard on remote procedure call.

Authorized licensed use limited to: IEEE Xplore. Downloaded on November 21, 2008 at 09:50 from IEEE Xplore. Restrictions apply.