Optimizing Ada On The Fly

Sheri J. Bernstein
AverStar, Inc.
23 Fourth Ave.
Burlington, Mass. 01803
(781) 221-6990
sheri@averstar.com

Robert A. Duff
AverStar, Inc.
23 Fourth Ave.
Burlington, Mass. 01803
(781) 221-6990
bobduff@averstar.com

1. ABSTRACT

One of the features that makes Ada such a reliable programming language is its use of run-time constraint checks. However, such checks slow program execution and increase program size.

The AdaMagic™ compiler uses one-pass, on-the-fly flow analysis during intermediate-language generation specifically to eliminate unnecessary constraint checks and to warn about potential error situations such as the use of uninitialized variables.

The one-pass nature of our method allows for faster compile times than more elaborate multi-pass flow analysis. Thus, we are able to generate efficient code without the impact on compile-time, and without the added implementation complexity, associated with a traditional separate optimization phase.

1.1 Keywords

Ada, compiler, optimization, check elimination, range checking, range propagation, array bounds checking, warnings, uninitialized variables.

2. INTRODUCTION

One of the features that makes Ada such a reliable programming language is its use of run-time constraint checks. However, there are two drawbacks with run-time checks:

1) **Efficiency**. Run-time checks slow the program's execution and increase its size. For an embedded real-time system, it is especially critical to minimize size and maximize run-time speed.
2) **Safety**. Although run-time checks increase the probability that testing will catch bugs, it is more desirable to catch potential problems sooner in the development cycle. In general, catching a bug earlier reduces the amount of time (and money) spent testing/debugging/rebuilding one's system.

The efficiency issue can be alleviated if the compiler can prove, at compile time, that certain run-time checks cannot fail. In such cases, the compiler can eliminate the code associated with the checks. The safety issue can be alleviated if the compiler can prove, at compile time, that certain checks will fail, and then issue warnings about these problems. In such cases, the programmer can fix these problems immediately, without any need for testing and debugging.

The programmer could use pragma Suppress to eliminate the overhead of run-time checks, if he can prove by hand that certain checks will not fail; he can even tell the compiler to blindly suppress all checks. However, if the programmer is mistaken, suppressing checks will often cause serious bugs, such as overwriting arbitrary memory locations. A safer solution is for the compiler to perform the proofs automatically, and to eliminate only those checks that cannot fail. This paper describes how the AdaMagic compiler uses one-pass, on-the-fly flow analysis to eliminate unnecessary checks, and to warn about potential error situations.

Most Ada compilers are able to eliminate some checks based on declarative information. However, to do a better job of check elimination, flow analysis is necessary. Consider the example in Figure 1. Ada requires an index check on A(I), but it's easy to eliminate this check based purely on the fact that I's subtype range matches the index subtype of Float_Array. However, in more complex cases such as the example in Figure 2, the relevant declarations are not enough. Our method allows the elimination of the index check on A(I). The compiler recognizes that, although the subtype of I allows out-of-range values, these values can never exist inside the if statement.

```
type Index is range 1..100;
type Float_Array is array(Index) of Float;

procedure P(A: Float_Array) is
   I : Index := Index'First;
begin
   ... A(I) ...
end P;
```

Figure 1. Check Elimination Uses Declarative Information

```
procedure P(A: Float_Array) is
   I: Integer;
begin
   ... -- some code that sets I
   if I in Index then
      ... A(I) ...
   end if;
end P;
```

Figure 2. Check Elimination Requires Flow Analysis

The AdaMagic technology consists of a Front End and a portable Ada run-time system, as well as various other tools. The technology is intended to be combined with a Back End that was more than likely designed to optimize and generate code for a language more like C or Pascal. The Front End parses the input, performs semantic analysis, and emits a fairly low-level intermediate language ("IL"), all in a (mostly) single pass. The Front End was designed to have a tailorable IL emitter so that it could directly generate whatever IL was needed by the Back End. For example, in the case of our "AdaMagic with C Intermediate" compiler, the Front End emits C source code, and the Back End consists of a standard C compiler. Our AppletMagic™ compiler generates Java byte codes as the intermediate language.

Because generation of IL occurs in the same pass as the rest of the Front End processing, constraint-check elimination is most conveniently implemented by performing on-the-fly "basic-block"-oriented flow analysis during IL generation, and therefore is target-independent. The Back End is presumed to take the IL and perform "traditional" optimizations like common subexpression (CSE) elimination, loop optimizations, etc.; as such, the primary goal of the Front End optimizer is to perform Ada-specific optimizations.

2. TRACKED INFORMATION

As each statement in an Ada subprogram is processed, the compiler checks to see if it can learn anything interesting about the value of local variables, formal parameters, and expression results. Value-related items of interest (collectively referred to as "value-info") are recorded using the data structure in Figure 3, which is associated with each variable and expression result being tracked. The fields of the data structure are described in more detail below.

```
type Value_Info is
   record
      Possible_Values : Value_Set;
         -- the set of values the variable could have
      Init_State : Init_State_Kind;
         -- known to be initialized?
      BB: Basic_Block_Ptr;
         -- the basic block in which this value
         -- information applies
      ... invalidation flags ...
         -- flags used to track Ada constructs that
         -- could invalidate value-info that has
         -- already been gathered
   end record;
```

Figure 3 : Data Structure To Record Value-Info

2.1 Possible Value Sets

For each local variable, the compiler tracks the set of possible values that the variable might have at any particular point in the program.

Possible value sets are represented by the (private) type Value_Set, which is an access type to Value_Set_Rec. The relevant data structures are supplied in Figure 4.

A set of integers is represented as a range plus a flag indicating whether the value 0 is included.

For example:
(Integer_Kind, Bounds => (-10, 10),
  Might_Be_Zero => True) represents -10..10.
(Integer_Kind, Bounds => (-10, 10),
  Might_Be_Zero => False) represents -10..-1 plus 1..10.
(Integer_Kind, Bounds => (1, 10),
  Might_Be_Zero => False) represents 1..10.

Note that this set type cannot represent all possible integer sets. For example, if the compiler can prove that a certain expression will always be 1, 2, or 4, then it must include 3 as well and pessimistically use 1..4.

Ranges are tracked because two of the most common constraint checks are range checks and index checks. The reason for tracking Might_Be_Zero is twofold. First, this helps us eliminate divide-by-zero checks. Second, we use the same integer sets to represent access values, with "null" being represented as zero. This helps us eliminate null checks.

```
  type UI_Ptr is ...;
  -- Represents a compile-time-known
  -- universal integer value.
  -- This is declared in a package that implements
  -- arbitrary-precision integer arithmetic, as is
  -- required for implementing static expressions.
  -- This type has special "infinite" values, which can
  -- be used to represent the range of all integers.

  type UI_Range is
    record
      First, Last: UI_Ptr;
    end record;
  -- Represents the range First..Last (inclusive).
  -- The empty range is always represented as 1..0.

  type Value_Set_Rec(Kind: Value_Set_Kind_Enum)
    is record
      case Kind is
        when Integer_Kind =>
          Might_Be_Zero: Boolean;
          Bounds: UI_Range;
        when Unknown_Kind =>
          null;
        when others =>
          ...
      end case;
    end record;
```

Figure 4 : Data Structures To Represent Possible Value Set

The possible values for an access-type expression are the actual run-time integer values used on the target machine. For example, on a typical 32-bit machine where null is represented as zero, the possible value set for an access-type expression is usually one of the following:

- 0..2**32 -- might be either null or non-null
- 0..0 -- known to be null
- 1..2**32 -- known to be non-null

Other sets are only possible in low-level code that uses address arithmetic and unchecked conversions; the compiler optimizes these accordingly. Similarly, type System.Address (whose full type declaration is an access type on our targets) is optimized in the same way; no special actions are needed. On machines with different addressing schemes, the representation is adjusted accordingly.

Enumeration types are represented in terms of integers, using the underlying integer code associated with each literal. Booleans are an important example of enumeration types, because the possible value set often allows us to know that some condition is always True or always False. The compiler can use this information to decide which side of an if statement is executed. Fixed point sets are also represented in terms of integers.

When the compiler cannot determine any information about a variable, it uses the Unknown_Value_Set to represent the variable's possible values : (Kind => Unknown_Kind).

In general, sets are allocated on the fly in a mark-release heap. However, certain sets which are commonly used, such as the empty set and the set of all integers, are preallocated and shared among all uses.

2.2 Init State

In Ada 95, reading an uninitialized scalar variable is a bounded error. To address this situation, our run-time model generates checks "early", which prevents the propagation of out-of-range values caused by uninitialized variables.

Consider the example in Figure 5.

```
   subtype S is Integer range 1..10;
  I, J : S;
  Message: String(S);

  ...
  I := J;
  while ... loop
     Message(I) := ...;
```

Figure 5 : Compiler Generates Constraint Check On Assignment

The compiler generates a constraint check on the assignment to I, to make sure that its value is within the range of subtype S. As such, there is no need for an index check on the array indexing. This early-checking model provides two important efficiency benefits:

1) Once initialized, a variable never becomes de-initialized. This is true whether the initialization occurs on the declaration or in a later assignment statement.
2) A formal parameter can never be uninitialized. This is because the compiler performs any necessary constraint checks on the actual parameters at the call site, guaranteeing that the formal parameters are within the range specified by their subtype.

Note that access variables are always initialized in Ada.

The init state is of interest to the compiler for the following reasons:

- An uninitialized variable can have a value outside of its declared range, so certain constraint checks cannot be eliminated.
- The compiler issues warnings for uses of possibly-uninitialized variables.

2.2.1 Representation Of Init State

The init states that a variable may have are identified in Figure 6.

```
  type Init_State_Kind is (
     Unknown,  -- compiler could not determine
               -- init state
     Initialized,  -- initialized (on all execution paths)
     Uninitialized, -- uninitialized (on all execution
                    -- paths)
     Not_Always_Initialized   -- uninitialized on at
                              -- least one execution
                              -- path
  );
```

Figure 6 : Type For Init States

2.2.2 Warnings For Possibly-Uninitialized Variables

The compiler issues an appropriate warning on the first use of an Uninitialized or Not_Always_Initialized variable. It doesn't warn the user if the init state of the variable is Unknown. Consider the example in Figure 7.

```
procedure Outer(I : Integer) is
   A, B, C, D: Integer;

   procedure Inner is
      Y: Integer;
   begin
      Y := A;       -- (1)
   End Inner;
begin
   A := B + I;      -- (2)
   Inner;
   if I > 3 then
      C := 10;      -- (3)
   end if;
   D := 3 * C;      -- (4)
End Outer;
```

Figure 7 : Variables With Different Init States

At (1), the init state of A is Unknown, since the compiler cannot determine if the up-level variable is initialized before the nested procedure is invoked. Because the compiler doesn't know if A is initialized, it doesn't issue a warning. At (2), the compiler detects that B is Uninitialized, and warns the user accordingly. At (4), C is initialized only if (3) is executed; otherwise, it is uninitialized. The compiler warns the user that C is Not_Always_Initialized.

The basic scheme for when the compiler issues a warning is as follows: If there is a basic block (see below) in the program where a variable is known to be in a certain state, and that basic block feeds a place where the state would result in an error, then a warning is appropriate, since there is a clear place in the program where a change could be made that would prevent the error. We consider this a case of "not always safe". By contrast, if no such basic block exists, a warning is not appropriate (even if we can't prove the situation is always safe), and the state is considered simply Unknown.

2.2.3 Initialized Variables

Our run-time model guarantees that the value of an initialized variable is within the range of its declared subtype. This allows the compiler to consult a variable's init state before emitting a constraint check, and to omit the check for an initialized variable if it can determine that the subtype's range is within the bounds of the check.

Knowing that a variable is initialized is also beneficial when the compiler must invalidate value-info that it has already gathered for a variable. This topic is discussed in more detail in section 3.6, **Invalidating Events**.

2.3 Basic Blocks

A basic block is a straight-line sequence of code with no flow of control [8]. Consider the example in Figure 8.

```
   X := X + 1;     -- bb1
   if X > 10 then
      return;      -- bb2
   else
      X := 2*X;    -- bb3
   end if;         -- bb4
   Y := X;
```

Figure 8 : Basic Blocks Identified In A Code Sequence

Basic-block-1 consists of the first statement, up through the evaluation of "X > 10". Basic-block-2 contains the body of the "then" clause, while basic-block-3 contains the body of the "else" clause. Basic-block-4 begins after the "end if". The naming convention "bb<n>" is used to identify the <n>th basic block.

If control can flow from one basic block to another, then the first block is said to "feed" the second. Thus, bb1 feeds both bb2 and bb3. bb4 is fed by bb3, but not by bb2 (since it returns from the subprogram).

A variable's value-info contains a reference to the basic block in which the value information applies.

There is always a "current" basic block; a new basic block is created (and made current) on the fly when we see a statement that starts a new basic block.

2.3.1 Unknown Basic Block

When the compiler is unable to determine all possible execution paths into a basic block, it represents this uncertainty in the flow graph with an "unknown basic block", which feeds into the basic block. An unknown basic block cannot de-initialize a variable, but it does pessimize the possible values that a variable could have.

The most common Ada construct that requires an unknown basic block is a loop, to represent the unknown effects of the loop iteration. This uncertainty exists because our approach constructs the flow graph and performs the optimizations in the same pass. A multi-pass approach would be able to iterate over the entire flow graph to better detect the effects of the iteration.

Consider the example in Figure 9.

```
procedure P is
   subtype S1_20 is Integer range 1..20;
   subtype S1_10 is Integer range 1..10;
   X : S1_20;
   Y : S1_10;
   type Array_Type is array(S1_10) of Integer;
   Array_Obj : Array_Type;
begin
   X := 3;                        -- (1)
   Y := 1;
   while Y < 10 loop
      ...Array_Obj(X)...          -- (2)
      Y := Y + 1;
      X := 20;
   end loop;
end P;
```

Figure 9 : While-loop Requires "Unknown Basic Block" Feeder

Because of the unknown basic block feeder for the while loop, the compiler is unable to precisely track the range of possible values for X. It does know that X has been initialized, so it uses declarative information to come up with a range of 1..20. As such, it has to perform the index check at (2). It would be wrong to omit this check, because the second loop iteration should raise Constraint_Error.

Note that the compiler also uses the unknown basic block to indicate the uncertainty of the code that is executed before labels, exception handlers, and nested subprograms.

3. INFORMATION GATHERING

When the compiler learns something interesting about the value of a local variable, it associates that information with the current basic block. First, it checks to see if the variable already has value-info that applies to the current basic block. If so, it updates the existing value-info with the new information. Otherwise, it creates a new value-info structure.

A local variable will be associated with at least one value-info structure (set up by the compiler at its declaration), but might have several, each with a different basic-block reference.

The following sections describe the situations in which the compiler gathers value-info.

3.1 Assignment Statements

When a local variable is the target of an assignment statement, its init state is set to Initialized. This occurs regardless of the right-hand side's init state; if the right-hand side is not in range at run time, the check that the compiler emitted for the assignment will raise Constraint_Error. The possible values from the right-hand side (after the check, if any) are copied into the target's value-info.

Consider the example in Figure 10.

```
procedure P(Y, Z : Integer) is
   A : Integer range 1..3;
   B : Integer range 7..9;
   X : Integer;
   type Array_Type is array(Integer range 1..15) of
     Integer;
   Array_Obj : Array_Type;
begin
   A := Y;
   B := Z;
   X := A + B;              -- (1)
   Array_Obj(X) := 3;       -- (2)
End P;
```

Figure 10 : Assignment Statement Provides Value-Info

At (1), the compiler evaluates the sum A + B and determines that the expression result is in the range 8..12 (details of this computation are provided below). The assignment copies this range to the set of possible values for X. With this knowledge, the compiler is able to eliminate the index check at (2), since it can determine that X is within the bounds of the array.

Note that if the right-hand side of the assignment statement isn't something that the compiler tracks (such as a record component), then the range of possible values for the target is set to the range of its subtype.

3.2 Expression Evaluation

We track the following kinds of expressions, for discrete types:

- name of a local variable

- numeric and enumeration literals
- some attributes, such as T'Min, T'Succ, on discretes
- binary + - * / ** mod rem, on integers (signed and modular)
- unary + - abs, on integers (signed and modular)
- and-then or-else and or xor not , on modulars and booleans
- comparisons on discretes

In general, the possible value set for each expression is determined from the possible value sets of subexpressions. For each predefined operator, the value-set package exports a corresponding set operation, which calculates the possible value set for that operator. For example, Set_Plus(X, Y) returns a set containing every value that is the sum of some value in X and some value in Y. For the result set, Might_Be_Zero is True if zero is in the new range.

In Figure 10, the possible value set for A is 1..3; for B, it is 7..9. The Set_Plus operation for A and B yields the set 8..12, with Might_Be_Zero equal to False.

The Set_Less_Than operation will typically return 0..1 (representing False..True), but in some cases it can do better. If the sets are disjoint, we can know that the result is either definitely True or definitely False.

3.3 Conditional Statements

This section explains how we represent information gathered from conditional statements, such as if statements and while loops. Consider the example in Figure 11.

```
while List /= null loop
   <lots of references to List.all>
    List := List.all.Next;
end loop;
```

Figure 11 : Conditional Statement Provides Value-Info

Inside the loop, we know that List is not null, and after the loop, we know that List is null. Therefore, we omit null checks for List.all inside the loop. If List.all occurs after the loop, we warn that it will fail.

Figure 12 shows a more complicated example.

```
if (not (X = null)) and (Y in Natural) then
   <then part>
else
   <else part>
end if;
```

Figure 12 : Conditional Statement Provides Value-Info

In the <then part>, we know that X is non-null and that Y is greater than or equal to zero. In the else part, we don't know anything about X or Y (based on the condition), since we don't know which half of the "and" made it False.

In order to gather information about conditionals, we have two basic blocks (in addition to the current one), as illustrated by the type definition in Figure 13.

```
type Cond_Info is
   record
      If_False_BB: Basic_Block_Ptr;
      If_True_BB: Basic_Block_Ptr;
   end record;
```

Figure 13 : Data Structure For Basic Blocks Associated With Condition

Cond_Info is used only when evaluating a condition. The If_True_BB represents the information that is known if the current expression ends up being True at run time. Likewise, If_False_BB represents the information that is known if the current expression ends up being False at run time.

The If_True_BB is then used as a feeder to the basic block representing the place we jump when the condition is True. Likewise, the If_False_BB is used as a feeder into the basic block representing the place we jump when the condition is False.

For each kind of boolean expression, we may modify the values of Cond_Info, or we may attach information to these basic blocks.

Some examples are:

- **not X**: Interchange If_True_BB and If_False_BB while emitting X. The idea is that whatever we know if "X" turns out to be False, we should also know if "not X" turns out to be True, and vice-versa.

- **X and Y**: Keep If_True_BB the same, and set If_False_BB to null, while emitting X and emitting Y. The idea is that if the result of "X and Y" turns out to be True, we know that X is True and we know that Y is True. However, if the result of "X and Y" turns out to be False, we don't know whether X is True or False; likewise for Y.

- **X < Y**: For discrete and fixed-point types, note in If_True_BB that X's upper bound is one less than Y's upper bound, and vice versa, unless we already know more about X or Y. Note the opposite in If_False_BB.

We optimize case statements in a similar manner, but the mechanisms are simpler, because all of the values involved are static.

3.4 Merging

When the compiler needs the value of a variable that it has tracked, it looks for value-info associated with the current basic block. If it doesn't exist, then the compiler must deduce the information based upon the variable's value in basic block(s) that feed into the current basic block. This process may require merging value-infos that represent information in mutually-exclusive execution paths. Once the compiler has performed this process, it associates the new value-info (applicable to the current basic block) with the variable. Because the information has been calculated and is now readily available, the compiler doesn't have to repeat the deduction the next time that it retrieves the variable's value-info. This is essentially a "lazy merging" approach, where we perform value-set merging only if (and when) the merged value set is needed.

Consider the example in Figure 14.

```
procedure P(J : Integer) is
   X, Y, Z : Integer;
   type Array_Type is array(Integer range 1..10) of
      Integer;
   Array_Obj : Array_Type;
begin
   if J = 2 then            -- bb1
      X := 1;               -- bb2
      Y := 4;
   else
      X := 2;               -- bb3
      Y := 5;
   end if;
   Z := X + Y;              -- bb4   (1)
   Array_Obj(Z) := 2;       -- (2)
end;
```

Figure 14 : Merging Value-Infos From Different Basic Blocks

At (1), the compiler needs the value-info for X and Y that is associated with bb4, in order to produce value-info for the sum. The compiler has not stored value-info for these variables in bb4, so it looks for this information associated with the basic blocks that feed bb4 (namely, bb2 and bb3). Value-info for X was stored for both bb2 and bb3. The merging process performs a union of the possible value sets, which in this case yields 1..2. Likewise, value-info for Y was stored for both bb2 and bb3; the merge yields the set 4..5. The compiler calculates the possible value set for the sum X + Y to be 5..8; this set is transferred to Z by the assignment. At (2), the compiler is able to eliminate the index check, since it knows that Z is within the array bounds 1..10.

3.5 Constraint Check

After the compiler emits a constraint check for a variable, it can assume that the condition is true; if this were not the case, then Constraint_Error would be raised at run time. The set of possible values associated with the condition is copied into the variable's value-info.

Consider the example in Figure 15.

```
procedure P is
   X : Rec_Ptr; -- access type pointing to a record
begin
   ...
   X.A.all := 5; -- (1)
   X.B.all := 7; -- (2)
End P;
```

Figure 15 : Constraint Check Provides Value-Info

At (1), the compiler is unable to determine if X is null, so it emits a null check. After the check, it recognizes that X cannot be null, and modifies X's value-info accordingly. At (2), the compiler sees that X is non-null, and is able to omit the null check.

3.6 Invalidating Events

There are certain Ada constructs that force the compiler to invalidate the value-info that it has already gathered for a local variable. This is because these constructs could modify the variable, and the compiler is unable to determine the new value. In these situations, the compiler must forget what it has learned about the variable by resetting its value-info to reflect an unknown state (Init_State => Unknown, Possible_Values => Unknown_Value_Set).

For an initialized variable, however, the compiler doesn't have to fully pessimize the value-info. The run-time model insures that once a variable is initialized, it cannot be de-initialized; this implies that the value of an initialized variable will be within the range of its declared subtype. Therefore, invalidating the value-info of an initialized variable does not change its init state, and sets its possible values to the range of its subtype. Retaining this useful data, rather than resetting things to an unknown state, provides the compiler with more opportunities to eliminate unnecessary checks.

Two of the invalidating events that the compiler recognizes are discussed below.

3.6.1 Call To Nested Procedure

A call to a nested procedure could modify the value of a local variable via an up-level reference.

The compiler maintains a global count of calls to nested procedures, which it increments when such a call is seen. When storing value-info for a local variable, the compiler also stores the value of this global count (i.e., how many calls to nested procedures have been seen at the time that the value-info applies). When the compiler retrieves the value-info, it compares the value-info's call-count with the global call-count. If the global count is bigger, it means that a call to a nested procedure has occurred since the variable's value-info was set. The call could have modified the local variable, so its value-info is no longer trustworthy and must be invalidated.

Consider the example in Figure 16.

```
procedure Outer is
  X : Integer;
  Y : Integer;
  Z : Integer;

  procedure Inner(A : Integer) is
  begin
     Y := A * 2;
  end Inner;

begin
   X := 3;
   Y := 2;          -- (1)
   Inner(10);       -- (2)
   Z := Y;          -- (3)
end Outer;
```

Figure 16: Call To Nested Procedure Invalidates Value-Info

In Outer, the compiler creates value-info for Y at (1), which includes a value of zero for the number of calls to nested procedures that have been seen so far. At (2), the compiler increments the global call-count to one. When retrieving the value-info for Y at (3), the compiler compares the value-info's call-count of zero with the global count of one. The global count is bigger, indicating that a call to a nested procedure has occurred since the value-info for Y was set. Because the optimizer does not currently keep track of whether Inner modifies Y, the compiler must assume that Y has been modified and therefore invalidates Y's value-info.

3.6.2 Store Through Access Variable

The compiler assumes that any local variable whose address can be taken could be modified by a store through an access variable. To track such an event, the compiler uses a strategy similar to the one described above to handle calls to nested procedures. The difference is that this check is made only for a local variable that is declared as aliased, or that has had 'Address applied to it.

The compiler uses a conservative approach; it doesn't compare the subtype of the local variable with the pointer's designated subtype. For example, an assignment through a floating-point access type cannot modify an integral variable, but the current algorithm would perform the invalidation under this circumstance. This, perhaps, is a minor issue, since aliased scalar variables are not very common. Nonetheless, the implementation could be expanded to use multiple counters to distinguish pointer stores of different types.

4. WARNINGS

In addition to the constraint-check elimination described above, the flow analysis performed by the compiler allows it to detect and warn about some potential error situations. We have already discussed the warnings that the compiler emits for uninitialized variables; other warnings are described below.

4.1 Possibly-Null Access Variables

Before emitting a null check for a local variable, the compiler examines that variable's value-info. If the variable is known to be non-null, the compiler eliminates the check. If the variable is known to be null on all execution paths, or on at least one path, the compiler will issue an appropriate warning. The compiler will not emit a warning if it is unable to determine this information.

In Section 2.2.2, *Warnings For Possibly-Uninitialized Variables*, we described the compiler's scheme for deciding when to emit warnings. This logic also applies for warnings about the "nullness" of an access variable. For example, if a procedure is passed a parameter of a named access type, it might be null. However, we consider its nullness state as "unknown," and no warnings are given on dereference. On the other hand, if an explicit test for nullness is made (e.g. "if P = null then ..."), but no action is then taken to ensure the value does not remain null, we consider the "nullness" state to be "not always safe", and give warnings on dereference.

4.2 Missing Return Statements

Using the information provided by the basic block flow graph, the compiler can determine if a function fails to return a value on some execution path. If so, it will warn the user. In the example in Figure 17, the compiler detects that the end statement of F is reachable via the execution path of bb4 => bb5. The compiler warns that the function might end without returning a value, in which case Program_Error will be raised.

```
function F(X : Integer) return Integer is
begin
   case X is                -- bb1
      when 2 =>             -- bb2
         return 16;
      when 3 =>             -- bb3
         return 24;
      when others =>        -- bb4
         Text_IO.Put_Line(
            "Unexpected value in function F");
   end case;                -- bb5
end F;
```

Figure 17 : Compiler Warns About Missing Return Statement

5. STATISTICS

To demonstrate the efficacy of the value tracking described above, we compiled the run-time system (~60K lines) for one of our products, both with the value-tracking optimizations and without. The results are displayed in Table 1.

	Non-Optimized	**Optimized**
null checks	2051	856
range-check compares*	2982	1848
zero checks	154	140

Table 1 : Statistics

*For the purpose of these statistics, one range check is treated as two range-check compares. This is done because the compiler is sometimes able to eliminate just the low half or the high half of a range check, and this is a way to incorporate the half-checks into the statistics.

In this exercise, the highest payoff is a 58% reduction in the number of null checks. The compiler was able to eliminate 38% of the range checks. Some zero checks were also eliminated.

Both the optimized and non-optimized versions eliminate checks based on declarative information. The differences shown are due to the additional checks eliminated by the flow analysis.

6. POSSIBLE FUTURE ENHANCEMENTS

Enhancing an optimizer is a never-ending task. The *Full Employment for Compiler Writers* Theorem ([2], p.411) states that it is always possible to create a yet-more-optimizing compiler. Hence, one always has to make tradeoffs when designing an optimizer. Thus far we have focused on what we believe are the highest-payoff Ada-specific optimizations consistent with our goal of keeping the Front End optimizer relatively simple and fast. However, there are some additional Ada-specific optimizations we have thought about as possible enhancements. We describe these in the following subsections.

6.1 Set Representation

Our representation of possible value sets is conservative. We could improve it in several ways. For example, we could generalize the Might_Be_Zero flag to a bit-map for small integer values, say, in the range -10..255, thus enabling us to represent sets like {1, 2, 4, 8, 27}. This would cover most enumeration types.

The limitation to this sort of extension is compile-time space: If we tried to represent all sets of integers, we would quickly fill up the compile-time address space with marginally useful information.

6.2 Objects Tracked

In addition to tracking local variables, parameters, and expression results, we could track global variables, heap objects, and components. It is not clear how high the payoff would be, because invalidating events would take their toll. For example, the compiler would need to assume that a call to an external procedure might affect any global variable.

We already track floating-point objects to the extent of determining their init state. However, we do not currently track floating-point ranges. The representation of such sets might require open ranges (e.g., “known to be greater than (but not equal to) zero”), whereas for integers, such a set is the same as “known to be >= 1”. Alternatively, it would require knowledge of the target machine, which is especially tricky for cross-compilers.

In any case, floating-point ranges are less important than discrete ranges, since the latter are used to index arrays.

6.3 Discriminant Checks

Although we do eliminate discriminant checks based on declarative information, we do not yet take advantage of flow analysis to eliminate them. We believe tracking possible values of discriminants is a straightforward extension of the above methods for discrete types.

6.4 Super-null Arrays

A null String in Ada typically has bounds 1..0. However, bizarre bounds like -100..-200 are allowed. Although rare, such ranges cause inefficient code. For example, the code to calculate the length of an array is not merely X'Last-X'First+1, but requires a conditional jump, which is expensive on modern machines. We have designed, but not yet implemented, an extension of the above method that could detect the usual case where null ranges are represented as T'First..T'First-1.

7. RELATED WORK

Basic-block-oriented data and control flow analysis has formed the foundation of most compiler optimizations for decades (e.g. see [1], [2], and [8]). Furthermore, there are a number of papers in the literature on range-check elimination based on multi-pass flow analysis (e.g. [3] and [4]). However, we were unable to locate any papers that mentioned our approach of one-pass, on-the-fly, “lazy-merging” flow analysis.

We chose this on-the-fly approach because it had the potential for providing significant reductions in the number of run-time checks, while requiring minimal changes to the structure of our existing Ada Front End and imposing relatively little compile-time overhead. The Back Ends that have been integrated with our Front End generally include high-quality “traditional” optimizers, but they do not do a particularly good job of eliminating run-time checks. One reason is that communicating all the subtype information through the IL is difficult and can bloat the IL. A second is that the propagation of range information is not very useful to an optimizer for languages like C that have little or no run-time-checking.

Much of the literature on range propagation is oriented toward array subscript checking, particularly in loops (e.g. [3], [4], and [5]). This is relevant for some “friendly” Fortran compilers. However few languages other than Ada have more general range checking concerns.

There have been a few papers specifically oriented toward optimizing Ada run-time checks (e.g. [7] and [9]). These identify very similar problems, and similar approaches to solutions, though they are generally multi-pass, “eager” merging approaches. Furthermore, little mention is made of the issue of providing helpful warnings, which we believe to be one of the most important features of the additional flow analysis being performed. We found no discussion of the issue of distinguishing between the totally “unknown” state, versus the “unsafe on some path” state that we believe is appropriate for a warning.

8. CONCLUSION

One-pass, on-the-fly, basic-block-oriented flow analysis can allow a compiler to gather much useful information about possible values and constraint checks. This allows the compiler to eliminate a surprising number of checks (compared to more global flow analysis). Furthermore, it allows the compiler to give useful warnings at compile time for errors that would otherwise go undetected until run-time, or worse, in the case of uninitialized variables, might cause unpredictable results.

One interesting additional benefit is that the optimized ANSI C that our “AdaMagic with C Intermediate” compiler produces is more readable. Instead of cluttering the generated C code with useless checks, many can be eliminated. This allows the C code to naturally reflect the original Ada code, to the point where directly debugging at the C source code level is feasible (though generally not necessary thanks to the use of #line directives and other tricks to enable Ada source level debugging using an otherwise C-oriented debugger).

The one-pass nature of our method allows for faster compile times than more elaborate multi-pass flow analysis. In addition, this method allows us to merge information from different basic blocks on the fly, one variable at a time. If information about a particular variable isn’t needed, we don't bother to deduce its value using data that might be available from earlier basic blocks. This also speeds compilation time.

Thus, we are able to generate efficient IL for Ada without the impact on compile time, and without the added implementation complexity, associated with a traditional separate optimization phase.

9. REFERENCES

[1] Aho, Alfred V., and Ullman, Jeffrey D. Principles of Compiler Design. Addison Wesley, 1977.

[2] Appel, Andrew W. Modern Compiler Implementation in Java. Cambridge University Press, 1998.

[3] Asuru, Jonathan M. Optimization of Array Subscript Range Checks. ACM Letters on Programming Languages and Systems, June 1992.

[4] Chin, Wei-Ngan and Goh, Eak-Khoon. A Reexamination of "Optimization of Array Subscript Range Checks". ACM Transactions on Programming Languages and Systems, March 1995.

[5] Kolte, Priyadarshan and Wolfe, Michael. Elimination of Redundant Array Subscript Range Checks. Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation, June 1995.

[6] Markstein, Victoria and Cocke, John and Markstein, Peter. Optimization of Range Checking. Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, June 1982.

[7] Møller, Peter Lützen. Run-Time Check Elimination for Ada 9X. Proceedings of the TRI-Ada '94 Conference & Expo, November 1994.

[8] Muchnick , Steve S. and Jones, Neil D. Program Flow Analysis -- Theory and Applications. Prentice-Hall, 1981.

[9] Schwarz, Birgit and Kirchgässner, Walter, and Landwehr, Rudolf. An Optimizer for Ada -- Design, Experiences and Results. Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation., June 1988.