Extended Abstract:

0. This Is An Extended Abstract

This is an extended abstract, and as such it does not contain the level of detail which the final paper will have. Topics mentioned herein will be expanded upon. Source code examples will be provided to illustrate points. Additional topics will be included, such as tracking local variables through code in which the flow of execution (e.g., loops, labels, exception handlers) cannot be sufficiently determined.

1. Introduction

One of the features which makes Ada such a reliable programming language is its use of run-time constraint checks. Such checks allow an Ada application to catch and handle Constraint_Error exceptions, thereby contributing to the program's robustness. However, these run-time checks can greatly increase the size and decrease the speed of a compiled Ada application. For an embedded real-time system, it is especially critical to minimize size and maximize run-time speed.

The user can direct the compiler to omit certain kinds of constraint-checks via pragma Suppress, but that can make an application vulnerable to specific categories of unhandled exceptions. A safer solution is to examine the Ada program during compilation, and omit only those checks which are determined to be unnecessary. Such an analysis relegates some of the expense of constraint-checking to compile-time, rather than run-time.

This paper describes Averstar's modification of its Ada-Magic front-end to eliminate unnecessary access, range, index, and overflow checks on local variables. It also describes how the front-end uses the information which is gathered and maintained for constraint-check elimination to warn about potential error situations.

2. Technical Summary

The constraint-check elimination is implemented by performing on-the-fly "basic-block"-oriented flow analysis during IL generation. Basically, as each line in an Ada subprogram is processed, the front-end checks to see if it can learn anything interesting about the value of a local variable. Value-related items of interest are hereafter referred to as "value-info". If the front-end can ascertain some value-info about a local variable in the statement, it creates/updates the optimization-information (hereafter referred to as "optim-info") associated with that variable. Both value-info and optim-info are described in the "Information Which Is Tracked For Local Variables" section which follows. Before a constraint-check on a local variable is emitted, the front-end checks the optim-info (if any) to determine if the check is unnecessary. If so, it does not generate the check.

However, it is not enough to keep track of the value-info associated with a local variable. It is also necessary to keep track of where in the Ada subprogram that value-info applies to the variable. For example, a local variable may be set to one value in the "then" arm of an "if" statement, and set to a different value in the "else" arm. This is relevant, since the modifications occur in mutually-exclusive execution paths. As such, the front-end needs to distinguish the value-info learned in the "then" arm from the value-info learned in the "else" arm. To solve this problem, the front-end breaks down an Ada subprogram into a sequence of "basic blocks". Each optim-info maintained

for a local variable contains a reference to a basic block, to identify the section of the subprogram in which the optim-info applies. The "Basic Block Usage" section describes how the front-end represents an Ada subprogram using basic blocks.

Therefore, a local variable being tracked by the front-end will have associated with it at least one optim-info, but possibly a list of them, indicating the variable's different "values" in different parts of the Ada subprogram. The front-end's use of the optim-info is described in the "Using Optim-info To Eliminate Constraints Checks" section.

There are times when the front-end must "forget" the value-info which it is tracking for a local variable. For example, the front-end recognizes that a nested procedure call could modify a local variable via an up-level reference, or that a store through an access variable could modify a local variable which has had its address taken. Another example is a procedure call, which invalidates the value-info of a local variable that is passed as an out parameter.

3. Information Which Is Tracked For Local Variables

As described in the "Technical Summary", this paper refers to the information which the front-end tracks for local variables as "optim-info". The value-related component of that information is referred to as "value-info"; it is the same as the "range-info" which is described below.

3.1 Range-Info

The front-end does not attempt to track the specific contents of a local variable. Rather, it tracks a range of possible values which the variable could have. This information is maintained for both scalar and access types.

3.2 State-Of-Initialization

"Initialization" refers to whether or not the variable has had an initial value set, either at declaration or via an assignment statement within the subprogram. Note that once a scalar variable is initialized, it doesn't become uninitialized. If a variable has not been initialized, then the only range information that the front-end knows about it is the range associated with its physical type.

3.3 Basic Block Reference

A variable's optim-info contains a reference to the basic block (i.e., section of the Ada subprogram) in which the value information applies.

3.4 Flags To Track Ada Constructs Which Invalidate Optim-info

As described in the "Technical Summary", both a nested procedure call and a store through an access variable could modify certain kinds of local variables. Thus, before trusting the optim-info of a local variable, the front-end makes sure that neither of these "invalidating events" have occurred since the variable's value-info was stored in the optim-info. If such an event has occurred, then the front-end cannot rely on the optim-info, and must invalidate it.

3.4.1 Nested Procedure Call

A nested procedure call could modify the value of a local variable via an up-level reference. To track nested procedure calls, the front-end maintains a global count for the number of nested calls seen; it increments this count whenever it sees a nested call. When the front-end creates/updates

optim-info for a local variable, it also stores the current value of the global count. When consulting the optim-info for a local variable, the front-end compares the optim-info's count with the current value of the global count. If the global count is larger, that means that a nested call occurred sometime after the local variable's optim-info was recorded. As such, that information is no longer trustworthy (and will be invalidated).

3.4.2 Store
Through An Access Variable The front-end assumes that any local variable whose address can be taken could be modified by a store through an access variable. The front-end uses the same strategy to track stores through an access variable as it does to track nested procedure calls. The difference is that this check is made only for a local variable which is declared as aliased, or which has had 'Address applied to it.

4. Basic Block Usage
As described in the "Technical Summary", it is necessary to correlate a local variable's value-info with the section of the Ada subprogram in which that value-info applies. To perform this correlation, the front-end breaks down an Ada subprogram into a series of "basic blocks". Basic blocks are chained together via "feeders" (internally represented as pointers to other basic blocks), which represent direct execution paths. For example, both the basic block which starts the "then" arm of an if-statement, and the basic block which starts the "else" arm, are "fed by" the basic block which contains the if-statement. The basic block which starts the "end-if" is typically "fed by" the last basic block in the "then" arm and the last basic block in the "else" arm (unless there is an explicit break in the execution flow, such as a return statement). Note that the front-end constructs the basic blocks on the fly during IL generation. This differs from a more widely-used approach, which creates the IL, then makes a separate pass to create the basic block graph, and then makes a separate pass to perform the optimizations. Clearly, the main advantage to performing the optimizations during IL generation is the huge compile-time savings. However, the one-pass solution does make it more difficult to track local variables in the presence of Ada constructs (e.g., gotos/labels) which would benefit from the more complete picture that a second pass would provide.

The following are the general rules which the front-end uses to break up an Ada subprogram into a sequence of basic blocks :
1) A "basic block" is a sequence of Ada statements which ends either with
   a compound statement (e.g., "if", "case", "while") OR ends because
   a new basic block starts.
2) A new basic block starts either because it is an arm of a compound
   statement (e.g., the "then" and "else" arms of an "if" statement),
   OR because more than one basic block "feeds" into it (such as the
   end-if in the above example).

(Additional rules exist; these are the most widely-applied ones.)

The internal representation of a basic block contains a unique numeric-id, which is allocated sequentially. This allows the front-end to make some assumptions about the order of basic blocks within the subprogram.

5. Learning New Value Information About A Local Variable
When the front-end can learn something interesting about the value of a local variable, it either updates an existing optim-info for the variable within the current basic block, or creates a new optim-info. When a new optim-info is created, it is added tothe optim-info list which isassociated with the variable.

The most intuitive example of new value information is an assignment statement into a local variable. Less obvious, but equally important, is the value information which can be learned from a boolean condition (such as in an "if" or "while" statement). For example, consider the following :

```
if x > 3 then
   y := x;
else
   z := x;
end if
```

In the "then" arm of the if-statement, the front-end creates an optim-info for "x" to indicate that its value is > 3 (i.e., its range is 4..<high>, where <high> is either some high boundary which has been tracked so far, or the high range of its physical type). Likewise, in the "else" arm, the front-end creates an optim-info for "x" to indicate that its value is <= 3.

6. Using Optim-info To Eliminate Constraint-Checks Before the front-end emits a constraint-check for a local variable (let's call it "var"), it uses the optim-info associated with "var" (if any) to determine if the check is unnecessary and can therefore be eliminated.

If "var" has optim-info which is associated with the current basic block, the front-end can easily use its contents to determine the necessity of the constraint-check. Otherwise, the front-end has to deduce "var"'s value-info for the current basic block. This is accomplished by "merging" those optim-infos associated with basic blocks that "feed" (i.e., have an execution path into) the current basic block. Once the front-end has performed this operation, it takes the optim-info which is the result of the merging, associates it with the current basic block, and adds it to "var"'s optim-info list. As a result, the front-end doesn't have to repeat the deduction which it has just performed.

7. Added Benefits Of Optim-Info And Basic Block Data In addition to constraint-check elimination, the front-end uses the optim-info and basic block data to issue user-friendly warnings about potential error situations. It uses the optim-info's "state of initialization" to warn that a local variable is either definitely uninitialized, or may be uninitialized. It uses the optim-info's range-info to warn that an access value is either definitely null, or may be null. It uses the basic block representation of a subprogram, which describes the flow of execution, to warn that a function may be missing a return statement.

8. Conclusion

This paper has given a general description of the constraint-check elimination which Averstar's Ada-Magic front-end now performs. Given the prevalence of constraint-checks in Ada, this implementation has yielded optimizations which are evident throughout the generated code.