# Eliminating Redundant Range Checks in GNAT Using Symbolic Evaluation

Johann Blieberger and Bernd Burgstaller

Department of Computer-Aided Automation
Technical University Vienna
A-1040 Vienna, Austria
{blieb,bburg}@auto.tuwien.ac.at

**Abstract.** Implementation of a strongly typed language such as Ada95 requires range checks in the presence of array index expressions and assignment statements. Range checks that cannot be eliminated by the compiler must be executed at run-time, inducing execution time and code size overhead. In this work we propose a new approach for eliminating range checks that is based on symbolic evaluation. Type information provided by the underlying programming language is heavily exploited.

## 1 Introduction

Strongly typed programming languages impose non-trivial requirements on an implementation in order to ensure the properties of the type system as specified by the language semantics. For discrete types these properties are often expressed as range constraints that must be met by values of a given type. For example, [Ada95] requires for signed integer types that Constraint_Error is raised by the execution of an operation yielding a result that is outside the base range of the respective type. In the case of constrained subtypes the constraints imposed by the requested ranges have to be enforced. In the case of array access, [Ada95] requires that each index value belongs to the corresponding *index range* of the array.

Integrity of a given program with respect to type system properties is usually enforced through checks performed for each program statement that can potentially violate them. Every check that cannot be proved redundant at compile-time has to be postponed until run-time involving two additional comparison and conditional branch instructions that need to be executed in order to ensure that a given value is within a required range. The resulting impact both in execution time and code size is high enough to justify the existence of compile-time options and language constructs that allow (partial) suppression of checks but open up a potential backdoor to erroneous program execution in turn.

On the contrary, an approach that increases the number of checks that can be proved redundant at compile-time aids in avoiding this backdoor by reducing run-time overhead and thus supporting "acceptance" of those checks for which either no proof of redundancy can be found or no such proof exists at all.

```
1   subtype sint is integer range -10 .. +10;
2   ...
3   procedure Swap (u, v : in out sint) is
4   begin
5       u := {u+v}^R1;
6       v := {u−v}^R2;
7       u := {u−v}^R3;
8   end Swap;
```

| Interval Arithmetic | | Symbolic Values | |
|---|---|---|---|
| u | v | u | v |
| $[-10, +10]$ | $[-10, +10]$ | $\underline{u}$ | $\underline{v}$ |
| $[-20, +20]$ | $[-10, +10]$ | $\underline{u} + \underline{v}$ | $\underline{v}$ |
| $[-10, +10]$ | $[-20, +20]$ | $\underline{u} + \underline{v}$ | $\underline{u}$ |
| $[-20, +20]$ | $[-10, +10]$ | $\underline{v}$ | $\underline{u}$ |

**Fig. 1.** Swapping the contents of two variables

It is the goal of our range check elimination method to identify redundant range checks that cannot be spotted by existing state-of-the-art compiler technology.

The outline of this paper is as follows. In Section 2 we present the notions of the symbolic data flow analysis framework used. In Section 3 we demonstrate the effectiveness of our framework through an example. Section 4 is devoted to the implementation of our dataflow framework within GNAT. Section 5 presents the experimental results gathered so far. In Section 6 we survey related work. Finally we draw our conclusion in Section 7.

## 2 Symbolic Evaluation

### 2.1 Motivation

Consider the code fragment given in Fig. 1 which swaps the values stored in $u$ and $v$ in place. It contains standard Ada95 code except for the numbered curly braces "$\{\ldots\}^{R\text{<number>}}$" around those expressions for which GNAT will issue a range check (cf., for instance, line 5).

Fig. 1 shows the intervals that GNAT derives by the standard interval arithmetic ([Hag95]) of Fig. 2[‡]. Three range checks, one for each assignment, are necessary because the computed intervals of the expressions on the right hand side exceed the range of subtype *sint*.

$$
\begin{aligned}
&[a, b] + [c, d] = [a + c, b + d] \\
&[a, b] - [c, d] = [a - d, b - c] \\
&[a, b] * [c, d] = [\min(ac, ad, bc, bd), \max(ac, ad, bc, bd)] \\
&[a, b]/[c, d] = [a, b] * [1/d, 1/c], 0 \notin [c, d]
\end{aligned}
$$

**Fig. 2.** Interval Arithmetic

[‡] In fact GNAT applies simplified rules for $[a, b] * [c, d]$ and $[a, b]/[c, d]$.

In contrast, the symbolic values in Fig. 1 show that range checks *R2* and *R3* are redundant. In line 6 variable $v$ is assigned the value of the actual parameter of $u$, namely $\underline{u}$, and in line 7 $u$ is assigned $\underline{v}$. If one of $\underline{u}$ or $\underline{v}$ would not be in the range of subtype *sint*, the range checks due to the Ada95 parameter passing mechanism would raise exception *Constraint_Error* and control flow would not reach the body of procedure *Swap*. It is this property, denoted as $\underline{u}, \underline{v} \in [-10, +10]$, together with the fact that the values for $u$ and $v$ are computed symbolically, that lets us actually derive that range checks *R2* and *R3* are redundant.

As can be seen by this introductory example symbolic analysis derives tighter bounds for the ranges of expressions and thus reduces the number of necessary range checks. The underlying theory of this approach is sketched in the remaining part of this section.

## 2.2 Preliminaries

*Symbolic evaluation* is an advanced static program analysis in which symbolic expressions are used to denote the values of program variables and computations (cf. e.g. [CHT79]). A path condition describes the impact of the program's control flow onto the values of variables and the condition under which control flow reaches a given program point. In the past, symbolic evaluation has already been applied to the reaching definitions problem [BB98], to general worst-case execution time analysis [Bli02], to cache hit prediction [BFS00], to alias analysis [BBS99], to optimization problems of High-Performance Fortran [FS97], and to pointer analysis for detecting memory leaks [SBF00].

The underlying program representation for symbolic evaluation is the *control flow graph (CFG)*, a directed labeled graph. Its nodes are basic blocks containing the program statements, whereas its edges represent transfers of control between basic blocks. Each edge of the CFG is assigned a condition which must evaluate to true for the program's control flow to follow this edge. *Entry (e)* and *Exit (x)* are distinguished nodes used to denote the start and terminal node.

In the center of our symbolic analysis is the *program context*, which includes *states* $\mathcal{S}_i$ and *path conditions* $p_i$. A program context completely describes the variable bindings at a specific program point together with the associated path conditions and is defined as

$$\bigcup_{i=1}^{k} [\mathcal{S}_i, p_i], \tag{1}$$

where $k$ denotes the number of different program states. State $\mathcal{S}$ is represented by a set of pairs $\{(v_1, e_1), \ldots, (v_m, e_m)\}$ where $v_s$ is a program variable, and $e_s$ is a symbolic expression describing the value of $v_s$ for $1 \leq s \leq m$. For each variable $v_s$ there exists exactly one pair $(v_s, e_s)$ in $\mathcal{S}$. A path condition $p_i$ specifies a condition that is valid for a given state $\mathcal{S}_i$ at a certain program point.

*Example 1.* The context

$$[\{(x, \underline{n}^2 - 1), (y, \underline{n} + 3)\}, x = y - 2] \tag{2}$$

consists of two variables $x$ and $y$ with symbolic values $\underline{n}^2 - 1$ and $\underline{n} + 3$, respectively. Variable $\underline{n}$ denotes some user input. The path condition ensures that $x = y - 2$ at the program point where the context is valid.

## 2.3 Type System Information and Range Checks

Informally a type $\overline{\mathcal{T}}$ is characterized by a set $\mathcal{T}$ of values and a set of primitive operations. To denote the type of an entity $j$ we write $\overline{\mathcal{T}}(j)$, to denote the set of values for $\overline{\mathcal{T}}(j)$ we write $\mathcal{T}(j)$. The fact that a given value $e \in \mathbb{Z}$ is contained in the set $\mathcal{T}(j)$ of a (constrained) integer type[§] is written as

$$e \in \mathcal{T}(j) \Leftrightarrow e \in \left[\overline{\mathcal{T}}(j)\text{'First}, \overline{\mathcal{T}}(j)\text{'Last}\right] \tag{3}$$

where $\overline{\mathcal{T}}(j)$'First and $\overline{\mathcal{T}}(j)$'Last denote the language-defined attributes for scalar types (cf. [Ada95]). A range check is a test of Equation (3) and is denoted as

$$e \in^? \mathcal{T}(j)\,. \tag{4}$$

In general the test is not performed on a value $e$ but on an expression $E$. Predicate *val* evaluates range checks symbolically within program contexts, which means that the check $E \in^? \mathcal{T}(j)$ is evaluated for each pair $[\mathcal{S}_i, p_i]$:

$$\text{val}(E \in^? \mathcal{T}(j), [\mathcal{S}_1, p_1] \cup \cdots \cup [\mathcal{S}_k, p_k]) \mapsto$$
$$[\mathcal{S}_1, p_1 \wedge \text{val}(E \in^? \mathcal{T}(j), [\mathcal{S}_1, p_1])] \cup \cdots \cup [\mathcal{S}_k, p_k \wedge \text{val}(E \in^? \mathcal{T}(j), [\mathcal{S}_k, p_k])].$$

This evaluation involves the symbolic evaluation of $E$ for $[\mathcal{S}_i, p_i]$, denoted as $\text{val}(E, [\mathcal{S}_i, p_i])$. If the result of this evaluation can be shown to be contained in the set $\mathcal{T}(j)$, then the evaluation of check $E \in^? \mathcal{T}(j)$ yields *true*, which means that the check is *redundant* for $[\mathcal{S}_i, p_i]$:

$$\text{val}(E \in^? \mathcal{T}(j), [\mathcal{S}_i, p_i]) = \begin{cases} \mathit{true} \Leftrightarrow \text{val}(E, [\mathcal{S}_i, p_i]) \in \mathcal{T}(j) \\ \mathit{false} \text{ else.} \end{cases} \tag{5}$$

Otherwise the range check is *required* or cannot be proved to be redundant. Deciding on the truth value of the above equation represents the center-piece of our range check elimination method. It depends on the data flow framework presented in Section 2.4, its exact treatment is thus deferred until Section 2.5.

Based on Equation (5) we define the necessity of a range check at a given program context via predicate ?rc. It evaluates to *false* only iff the range check is *redundant* for every pair $[\mathcal{S}_i, p_i]$ of the program context:

$$?\text{rc}\Big(E \in^? \mathcal{T}(j), \bigcup_{i=1}^{k} [\mathcal{S}_i, p_i]\Big) = \begin{cases} \mathit{false} \Leftrightarrow \underset{1 \le i \le k}{\forall} \big(\text{val}(E \in^? \mathcal{T}(j), [\mathcal{S}_i, p_i]) = \mathit{true}\big) \\ \mathit{true} \text{ else.} \end{cases} \tag{6}$$

For the compiler backend we map predicate ?rc to the node-flag *Do_Range_Check* of GNAT's *abstract syntax tree* (cf. also Section 4).

[§] In Ada terms [Ada95] a combination of a type, a constraint on the values of the type, and certain specific attributes is called *subtype*.

## 2.4 A Data-Flow Framework for Symbolic Evaluation

We define the following set of equations for the symbolic evaluation framework:

$$\text{SymEval}(B_{\text{entry}}) = [\mathcal{S}_0, p_0]$$

where $\mathcal{S}_0$ denotes the initial state containing all variables which are assigned their initial values, and $p_0$ is true,

$$\text{SymEval}(B) = \bigcup_{B' \in \text{Preds}(B)} \text{PrpgtCond}(B', B, \text{SymEval}(B')) \mid \text{LocalEval}(B) \tag{7}$$

where $\text{LocalEval}(B) = \{(v_{i_1}, e_{i_1}), \ldots, (v_{i_m}, e_{i_m})\}$ denotes the symbolic evaluation local to basic block $B$. The variables that get a new value assigned in the basic block are denoted by $v_{i_1}, \ldots, v_{i_m}$. The new symbolic values are given by $e_{i_1}, \ldots, e_{i_m}$. The *propagated conditions* are defined by

$$\text{PrpgtCond}(B', B, \text{PC}) = \text{Cond}(B', B) \odot \text{PC},$$

where $\text{Cond}(B', B)$ denotes the condition assigned to the CFG-edge $(B', B)$. Denoting by PC a program context, the operation $\odot$ is defined as follows:

$$\text{Cond}(B', B) \odot \text{PC} = \text{Cond}(B', B) \odot [\mathcal{S}_1, p_1] \cup \cdots \cup [\mathcal{S}_k, p_k]$$

$$= [\mathcal{S}_1, \text{Cond}(B', B) \wedge p_1] \cup \cdots \cup [\mathcal{S}_k, \text{Cond}(B', B) \wedge p_k]$$

The following definition gives rewrite rules for the $\mid$ operator, which integrate local changes of a basic block into the program state and path conditions.

**Definition 1.** *The semantics of the* $\mid$ *operator is as follows:*

1. *We replace* $\{\ldots, (v, e_1), \ldots\} | \{\ldots, (v, e_2), \ldots\}$ *by* $\{\ldots, (v, e_2), \ldots\}$.
2. *Furthermore* $\{\ldots, (v_1, e_1), \ldots\} | \{\ldots, (v_2, e_2(v_1)), \ldots\}$, *where* $e(v)$ *is an expression involving variable* $v$, *is replaced by* $\{\ldots, (v_1, e_1), \ldots, (v_2, e_2(v_1)), \ldots\}$. *For the above situations it is important to apply the rules in the correct order, which is to elaborate the elements of the right set from left to right.*
3. *If a situation like* $[\{\ldots, (v, e), \ldots\}, C(\ldots, v, \ldots)]$ *is encountered during symbolic evaluation, we replace it with* $[\{\ldots, (v, e), \ldots\}, C(\ldots, e, \ldots)]$.

This data-flow framework has been introduced in [Bli02].

**Solving the Data-Flow Problem** We solve the equation system defining the data-flow problem using an elimination algorithm for data-flow analysis [Sre95]. It operates on the *DJ* graph (DJG), which essentially combines the control flow graph and its dominator tree into one structure. Node $n$ *dominates* node $m$, if every path of CFG edges from *Entry* to $m$ must go through $n$. Node $n$ is the *immediate dominator* of $m$ if $n \neq m$, $n$ dominates $m$, and $n$ does not dominate any other dominator of $m$. The *dominator tree* is a graph containing every node of the CFG, and for every node $m$ an edge from its immediate dominator $n$ to $m$.

The elimination algorithm given in [Sre95] consists of two phases. The first phase performs DJ graph reduction and variable substitution of the equation system until the DJ graph is reduced to its dominator tree. Cycles (e.g. due to loops) are treated by the *loop-breaking* rule [RP86], [Bli02]. After the first phase the equation at every node is expressed only in terms of its parent node in the dominator tree. After determining the solution for the equation of Node *Entry*, the second phase of the algorithm is concerned with propagation of this information in a top-down fashion on the dominator tree to compute the solution for the other nodes. Every node corresponds to a basic block of the program under investigation, and its solution is expressed in terms of a program context as stated by Equation (1). By its definition such a context describes the possible variable bindings valid at this program point, and in this way it provides the information required for the range check elimination decision of Section 2.5.

**Hierarchical Data-Flow Frameworks** Since range checks are part of certain programming language constructs such as array access or assignment statements, a program analysis method that incorporates these checks has to be aware of control flow occurring on intra-statement level. The abstraction level of intra-statement control flow is in the same order of magnitude lower than inter-statement control flow as assembly language is compared to high-level language code. It is not desirable to spend the complete analysis of a program on intra-statement level since one gets easily overwhelmed by the amount of detail and tends to loose the view of the "big picture". For this reason we introduce a two-level hierarchy in our data-flow framework where the two levels correspond to analysis incorporating intra- and inter-statement control flow. We avoid intra-statement control flow as much as possible, which means that it is only considered for statements for which the compiler inserts a range check.

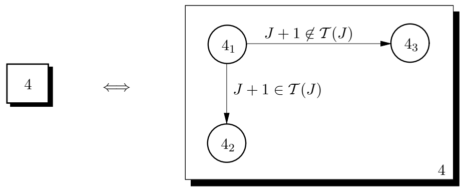

**Fig. 3.** Levels of Abstraction: Inter-Statement ⇔ Intra-Statement

As a notational convenience we collapse the intra-statement control flow subgraph into one single *compound* node of the inter-statement CFG. In this way Fig. 3 depicts the code associated with Node 4 of Fig. 4 as a collapsed compound node (left) and as a compound node expanded to its subgraph (right). Note that we use circular shapes for ordinary nodes and boxes for compound nodes to distinguish between the two.

### 2.5 The Range Check Elimination Decision

Consider Equation (8) which is an example of a valid symbolic equation according to the symbolic data-flow framework introduced in Section 2.4. Construction of an input program yielding this equation is straight-forward and suppressed for space considerations. The types of the used variables are $\overline{\mathcal{T}}(c) = \mathit{Boolean}$, and $\overline{\mathcal{T}}(n,x,y,z) = \mathit{Positive}$.

$$[\{(c,\bot),(n,\underline{n}),(x,\underline{x}),(y,\underline{y}),(z,\underline{z})\}, x^n + y^n = z^n] \mid \{(c, 3-n \overset{?}{\in} \mathcal{T}(x))\}. \quad (8)$$

The proposed range check can only be removed, if, according to Equation (5),

$$\mathrm{val}\big(3-n, [\{(c,\bot),(n,\underline{n}),(x,\underline{x}),(y,\underline{y}),(z,\underline{z})\}, x^n + y^n = z^n]\big) \in \mathcal{T}(x)\,.$$

This formula is valid if it it can be shown that $n \leq 2$, which requires a proof of

$$(\forall n)(\forall x)(\forall y)(\forall z)[x^n + y^n = z^n \Rightarrow n \leq 2],$$

also known as Fermat's last theorem. While for this specific theorem our data-flow framework could be aware of the recently discovered proof, in general there exists no algorithm capable of determining the validity of a formula such as Equation (5) stated in elementary arithmetic built up from $+$, $*$, $=$, constants, variables for nonnegative integers, quantifiers over nonnegative integers, and the sentential connectives $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$ subject to the requirement that every variable in such a formula be acted on by some quantifier. This follows from a conclusion from Gödel's incompleteness theorem, [Rog87, p. 38] contains the corresponding proof.

However, for a subclass of elementary arithmetic called *Presburger arithmetic*, validity is decidable [Sho79]. Presburger formulas are those formulas that can be constructed by combining first degree polynomial (*affine*) constraints on integer variables with the connectives $\neg$, $\wedge$, $\vee$, and the quantifiers $\forall$ and $\exists$. Constraints are affine due to the fact that Presburger arithmetic permits addition and the usual arithmetical relations $(<, \leq, >, \geq, =)$, but no arbitrary multiplication of variables[¶].

The Omega test [Pug92] is a widely used algorithm for testing the satisfiability of arbitrary Presburger formulas. We can use it as a range check elimination decision procedure if we are able to translate Equation (5) into such a formula. We split this translation into two steps, each yielding a conjunction $\Gamma$ of constraints. Initially we set $\Gamma_1 = \Gamma_2 = \mathit{true}$.

**Step 1:** We derive constraints from the path-condition $p_i$ of state $S_i$ as follows. The path-condition essentially is a conjunction of predicates $P_l$ that are *true* for state $S_i$:

$$p_i = P_1 \wedge \cdots \wedge P_N.$$

Each predicate $P_l$ corresponds to a CFG condition $C$ that is an expression involving program variables $v_c \in V_c$, where $V_c$ denotes the set $\{v_1, \ldots, v_m\}$ of possible

¶ Although it is convenient to use multiplication by *constants* as an abbreviation for repeated addition.

program variables (cf. also Equation (1)). This can be written as $P_l = C(V_c)$. Once we evaluate $C(V_c)$ for state $S_i$ (cf. Definition 1), we get $P_l = C(E_c)$ as a condition over symbolic expressions. Solving $C(E_c)$ yields the solution $L_l(V_c)$ for which $P_l = \mathit{true}$. Thus each predicate $P_l$ yields a constraint for $\Gamma_1$:

$$\Gamma_1 ::= \Gamma_1 \wedge \bigwedge_{1 \le l \le N} L_l(V_c). \tag{9}$$

*Example 2.* Starting from the context given in Equation (2), we have predicate $P_1 = C(V_c) : x = y - 2$, and $C(E_c) : \underline{n}^2 - 1 = \underline{n} + 1$ which yields the solutions $\underline{n}_1 = 2$, and $\underline{n}_2 = -1$ resulting in the constraint $(\underline{n} = 2 \vee \underline{n} = -1)$.

*Example 3.* Another example involves a predicate that arises from iteration-schemes of *for* and *while* loops. Context $[\{(x, 1), (y, 1), (z, \underline{z})\}, x \text{ in } y..z]$, yields the constraint $C(E_c) : 1 \le x \le \underline{z}$.

**Step 2:** While the previous step dealt with the information captured in the path condition $p_i$ of Equation (5), Step 2 addresses the translation of the proposed range check $\text{val}(E \in^? \mathcal{T}(j))$ into a conjunction $\Gamma_2$ of constraints. Like the conditions $C$ of Step 1, expression $E$ is a symbolic expression involving program variables $v_c \in V_c$. Again we evaluate $E(V_c)$ for state $S_i$ to get $E(E_c)$. We can now set up a constraint that requires $E(E_c)$ to be outside the range of type $\overline{\mathcal{T}}(j)$:

$$\Gamma_2 ::= \Gamma_2 \wedge j = E(E_c) \wedge \big(j < \overline{\mathcal{T}}(j)\text{'First } \vee j > \overline{\mathcal{T}}(j)\text{'Last}\big). \tag{10}$$

We then check by means of the Omega test whether there exists a solution satisfying the conjunction $\Gamma_1 \wedge \Gamma_2$. Non-existence of such a solution means that for $[\mathcal{S}_i, p_i]$ the expression $E(E_c)$ will be within the range of type $\overline{\mathcal{T}}(j)$. Completing Equation (5), we finally get

$$\text{val}(E \in^? \mathcal{T}(j), [\mathcal{S}_i, p_i]) = \begin{cases} \mathit{true} \Leftrightarrow \text{val}(E, [\mathcal{S}_i, p_i]) \in \mathcal{T}(j) \Leftrightarrow \Gamma_1 \wedge \Gamma_2 = \mathit{false} \\ \mathit{false} \text{ else.} \end{cases} \tag{11}$$

**Non-Affine Expressions and Conservative Approximations** A method capable of transforming certain classes of general polynomial constraints into a conjunction of affine constraints has been presented by Maslov [MP94].

For non-transformable nonaffine expressions that are part of (the solution of) a predicate $P_l$ (cf. Step 1 above), we can omit the constraint imposed by $P_l$ and hence generate a conservative approximation for $\Gamma_1$ for the following reason: given the set $\{v_1, \ldots, v_m\}$ of possible program variables, the mapping of the symbolic expression $E(E_c)$ to its value can be regarded as a function $f : \mathcal{T}(v_1) \times, \ldots, \times \mathcal{T}(v_m) \to \mathbb{Z}$. Each predicate $P_l$ potentially constrains the $m$-dimensional domain of $f$, which, depending on $E(E_c)$, potentially constrains the range of $f$. The smaller the range of $f$, the more likely it is that we can derive that the proposed range check of $E(E_c)$ is redundant. Omission of $P_l$ results in a conservative approximation of the range of $f$ in the sense that we might generate a *false positive* claiming that the proposed range check is needed whereas it is actually redundant (cf. also Equation (6)). False negatives are not possible since $P_l$ cannot *widen* the range of $f$.

## 3 Example

We demonstrate our range check elimination technique by an example for which it can be shown manually that no range check is needed. Therefore every language-defined range check is redundant and should be identified as such. Fig. 4 shows our example taken from the *Heapsort* algorithm as presented in [Sed88].

```
Max: constant Positive := ??;               -- Number of elements to be sorted
subtype Index is Positive range 1 .. Max;
type Sort_Array is array(Index) of Integer;
Arr : Sort_Array;

procedure Siftdown (N,K:Index) is
   J, H : Index;                                                 -- Node 1
   V : Integer;                                                  -- Node 1
begin
   V := Arr(K);                                                  -- Node 1
   H := K;                                                       -- Node 1
   while H in 1..N/2 loop                                        -- Node 2
      J := {2*H}^R1;                                             -- Node 3
      if J<N then                                                -- Node 3
         if Arr(J)<Arr({J+1}^R2) then                            -- Node 4
            J := {J+1}^R3;                                       -- Node 5
         end if;
      end if;
      if V >= Arr(J) then                                        -- Node 6
         Arr(H) := V;                                            -- Node 7
         exit;                                                   -- Node 7
      end if;
      Arr(H) := Arr(J);                                          -- Node 8
      Arr(J) := V;                                               -- Node 8
      H := J;                                                    -- Node 8
   end loop;
   return;                                                       -- Node 9
end Siftdown;
```

**Fig. 4.** Example: Procedure Siftdown

Fig. 5 shows the control flow graph of our example. It contains three compound nodes (3, 4, and 5) which correspond to the intra-statement analysis necessary to treat range checks $R1$, $R2$, and $R3$ (cf. Fig. 4). Compound Node 3 is expanded whereas compound Nodes 4 and 5 are collapsed due to space considerations. CFG edges are labelled with their associated conditions, edges without labels denote "*true*" conditions. The edge $e \rightarrow x$ is artificial in the sense that it is required by our elimination algorithm but does not correspond to "real" control flow. Symbol ↯ denotes CFG edges that correspond to control flow of a *Constraint_Error* exception taken due to failure of a range check. Since procedure *Siftdown* contains no exception handler and since we assume there is no calling

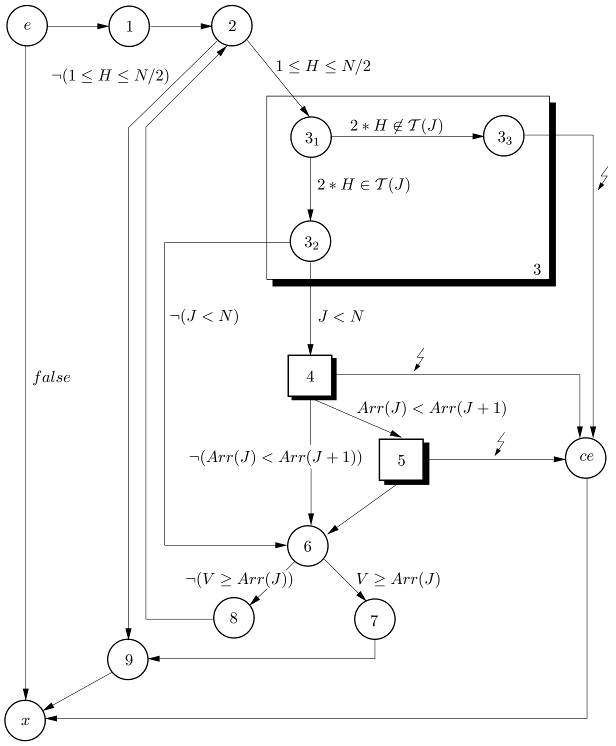

**Fig. 5.** Example: CFG of Procedure Siftdown

procedure with our example, these edges are simply "collected" by Node *ce* that is connected to the procedure's *exit* node.

Table 1 shows the set of SymEval equations for our example procedure. The symbol $\perp$ is used to denote undefined values. $p_1$ denotes the predicate $\underline{N} \in \mathcal{T}(N) \wedge \underline{K} \in \mathcal{T}(K)$ that is due to the parameter association mechanism of [Ada95]. Table 2 shows the sequence of elimination steps performed during the elimination phase in order to reduce the system of equations so that each equation depends only on its immediate dominator. For the purpose of our example it suffices to consider the result of the application of the loop-breaking rule at Step 23, where each induction variable is replaced by an (indirect) recursion:

$$2\ \not\emptyset\colon X_2 = X_1 \mid \{(Arr, Arr(\perp, \nu)), (H, H(\perp, \nu)), (J, J(\perp, \nu)), (C_{1\ldots7}, C_{1\ldots7}(\perp, \nu))\}.$$

For the purpose of our example it is furthermore sufficient to collapse all array assignments into one recursion $Arr(\perp, \nu)$.

In the propagation phase we propagate data flow information in a top-down manner on the dominator tree after the solution of the root node has been determined. Table 3 enumerates those steps for our example procedure.

For the sake of brevity we will focus on the examination of range check $R1$ located in Equation $X_{3_1}$. For this reason we are concerned with propagation

$$
\begin{aligned}
X_{\mathrm{e}} &= [\{(Arr, \underline{Arr}), (N, \underline{N}), (K, \underline{K}), (J, \bot), (V, \bot), (H, \bot), (C_{1\ldots7}, \bot)\}, p_1] \\
X_1 &= X_{\mathrm{e}} \mid \{(V, Arr(K)), (H, K)\} \\
X_2 &= (X_1 \cup X_8) \mid \{(C_1, (H \text{ in } 1 \mathinner{..} N/2))\} \\
X_{3_1} &= C_1 \odot X_2 \mid \{(C_2, (2 * H \in^? \mathcal{T}(J))\} \\
X_{3_2} &= C_2 \odot X_{3_1} \mid \{(J, 2 * H), (C_3, (J < N))\} \\
X_{3_3} &= \neg C_2 \odot X_{3_1} \mid \{(J, \bot)\} \\
X_{4_1} &= C_3 \odot X_{3_2} \mid \{(C_4, (J + 1 \in^? \mathcal{T}(Index))\} \\
X_{4_2} &= C_4 \odot X_{4_1} \mid \{(C_5, (Arr(J) < (Arr(J + 1))))\} \\
X_{4_3} &= \neg C_4 \odot X_{4_1} \\
X_{5_1} &= C_5 \odot X_{4_2} \mid \{(C_6, (J + 1 \in^? \mathcal{T}(J))\} \\
X_{5_2} &= C_6 \odot X_{5_1} \mid \{(J, J + 1)\} \\
X_{5_3} &= \neg C_6 \odot X_{5_1} \mid \{(J, \bot)\} \\
X_6 &= \big(\neg C_3 \odot X_{3_2} \cup \neg C5 \odot X_{4_2} \cup X_5\big) \mid \{(C_7, (V \geq Arr(J))\} \\
X_7 &= C_7 \odot X_6 \mid \{(Arr(H), V)\} \\
X_8 &= \neg C_7 \odot X_6 \mid \{(Arr(\{H\}), Arr(J)), (Arr(J), V), (H, J)\} \\
X_9 &= \neg C_1 \odot X_2 \cup X_7 \\
X_{ce} &= X_{3_3} \cup X_{4_3} \cup X_{5_3} \\
X_{\mathrm{x}} &= X_9 \cup X_{ce}
\end{aligned}
$$

**Table 1.** Set of SymEval Equations for Example *Siftdown*

steps 2, 3, and 4 of Table 3:
$e \to 1, 1 \to 2, 2 \to 3_1$

$$
\begin{aligned}
X_{3_1} = [\{ \quad & (Arr, Arr(\underline{Arr}, \nu)), (N, \underline{N}), (K, \underline{K}), (H, H(\underline{K}, \nu)), (J, J(\bot, \nu)), \\
& (C_{1\ldots7}, C_{1\ldots7}(\bot, \nu))\}, p_1 \wedge H \text{ in } 1 \mathinner{..} N/2] \mid \{(C_2, (2 * H \in^? \mathcal{T}(J))\}.
\end{aligned}
$$

In order to compute the solution for Node $X_{3_1}$, we evaluate predicate *val* according to Equation (5):

$$
\begin{aligned}
\mathrm{val}\big(2 * H \in^? \mathcal{T}(J), &[\{(Arr, Arr(\underline{Arr}, \nu)), (N, \underline{N}), (K, \underline{K}), (H, H(\underline{K}, \nu)), \\
& (J, J(\bot, \nu)), (C_{1\ldots7}, C_{1\ldots7}(\bot, \nu))\}, p_1 \wedge H \text{ in } 1 \mathinner{..} N/2]\big).
\end{aligned}
\tag{12}
$$

| | | | | |
|---|---|---|---|---|
| 1.) $5_3 \to ce$ | 6.) $7 \to 9$ | 11.) $4_1 \to 6$ | 16.) $3_2 \to 2$ | 21.) $3_1 \to x$ |
| 2.) $5_2 \to 12$ | 7.) $4_3 \to ce$ | 12.) $6 \to 2$ | 17.) $3_2 \to 9$ | 22.) $9 \to x$ |
| 3.) $5_1 \to ce$ | 8.) $4_2 \to 6$ | 13.) $6 \to 9$ | 18.) $ce \to x$ | 23.) $2 \emptyset$ |
| 4.) $5_1 \to 6$ | 9.) $4_2 \to ce$ | 14.) $3_3 \to ce$ | 19.) $3_1 \to 2$ | 24.) $2 \to x$ |
| 5.) $8 \to 2$ | 10.) $4_1 \to ce$ | 15.) $3_2 \to ce$ | 20.) $3_1 \to 9$ | 25.) $1 \to x$ |

**Table 2.** Elimination: from DJ-Graph to Dominator Tree

| | | | | | |
|---|---|---|---|---|---|
| 1.) $e \to x$ | 4.) $2 \to 3_1$ | 7.) $3_2 \to 4_1$ | 10.) $4_2 \to 5_1$ | 13.) $3_2 \to 6$ | 16.) $3_1 \to ce$ |
| 2.) $e \to 1$ | 5.) $3_1 \to 3_3$ | 8.) $4_1 \to 4_3$ | 11.) $5_1 \to 5_3$ | 14.) $6 \to 7$ | 17.) $2 \to 9$ |
| 3.) $1 \to 2$ | 6.) $3_1 \to 3_2$ | 9.) $4_1 \to 4_2$ | 12.) $5_1 \to 5_2$ | 15.) $6 \to 8$ | |

**Table 3.** Propagation of the DFA-Solution

Range check $R1$ is redundant if Equation (11) evaluates to *true*. From Equation (9) we get

$$\Gamma_1 ::= true \wedge 1 \leq \underline{N} \leq Max \wedge 1 \leq \underline{K} \leq Max \wedge 1 \leq H \leq \underline{N}/2,$$

where "/" denotes integer division and where the second conjunct is due to predicate $p_1$ and the third conjunct is due to predicate "$H$ in $1..N/2$". From Equation (10) we get

$$\Gamma_2 ::= true \wedge J = 2 * H \wedge \big(J < 1 \vee J > Max\big).$$

As expected the Omega test confirms that $\Gamma_1 \wedge \Gamma_2 = false$ which, according to Equation (11), means that the range check $2 * H \in^? \mathcal{T}(J)$ is redundant. Similarly we can derive at propagation steps 7 and 10 that range checks $R2$ and $R3$ are redundant.

## 4 Implementation

Fig. 6 is an extension of the structure of the GNAT compiler as explained in [SB94]. Our data-flow framework is situated between the frontend and the backend of the GNAT compiler. Its input is an abstract syntax tree ($AST$) that has been syntactically and semantically analyzed and expanded (cf. [SB94]).

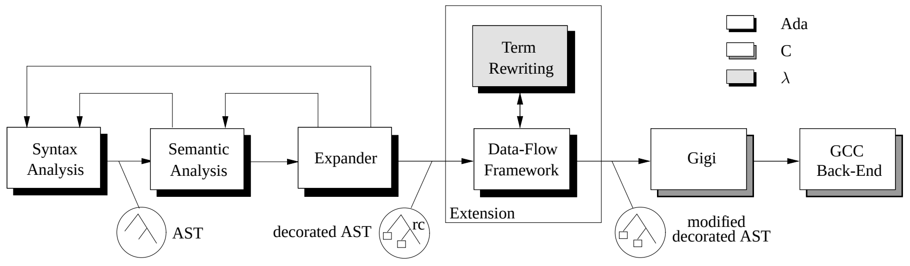

**Fig. 6.** Integration of the Data-Flow Framework Within GNAT

Every tree expression for which node flag *Do_Range_Check* is set requires the backend to generate a corresponding run-time check. The frontend makes use of this signaling mechanism for each range check that cannot be proved redundant at compile-time. When we build the CFG from the AST we generate an intra-statement control flow subgraph for every statement containing such an expression.

Once the CFG is extended by immediate domination edges, it is passed to the elimination algorithm which determines the sequence of equation insertion and loop-breaking steps in order to transform the DJG to its dominator tree (cf. e.g. Table 2). Equation insertion and loop-breaking itself is handled by the *term rewriting* component which, due to the nature of its requirements, depends on pattern matching and expression transformation capabilities available mainly in functional programming languages. An early prototype of this component has already been implemented as a *Mathematica* package [Mae90].

In order to compute the data-flow solution for each CFG node the elimination algorithm instructs the term rewriting component to perform the necessary propagation steps (cf. e.g. Table 3). If we can derive at a given CFG node that according to Equation (6) a given range check is redundant, we reset the AST flag *Do_Range_Check* which prevents the backend from generating the corresponding run-time check.

## 5 Experimental Results

In order to demonstrate the effectiveness of our approach we have considered several examples for which it can be manually proved that no range checks are necessary. The following programs have been examined so far: "Siftdown" (cf. Fig. 4) and the corresponding driver program "Heapsort" (cf. [Sed88]), "Mergesort", and "Quicksort". Table 4 compares the number of range checks required by the GNAT frontend to the number of checks that remain after symbolic analysis of the example programs. The number of assembler statements and the stripped object code size of the resulting executables are also given.

| | Post-GNAT | | | Post-Symbolic Evaluation | | |
|---|---|---|---|---|---|---|
| Source | Range-Checks | Asm-Stmts | Obj.-Size | Range-Checks | Asm-Stmts | Obj.-Size |
| Siftdown | 3 | 89 | 800 | 0 | 55 | 604 |
| Heapsort | 2 | 73 | 736 | 0 | 51 | 596 |
| Quicksort | 4 | 122 | 908 | 2 | 100 | 824 |
| Mergesort | 6 | 183 | 1128 | 0 | 119 | 812 |

**Table 4.** Experimental Results

For procedure "Siftdown" we have also measured the execution time overhead for range checks. We have found that the removal of superfluous range checks yields on average a performance increase of 11.7 percent, whereas the decrease in object code size is more than 24 percent. The performance figures given in Fig. 7 have been obtained on an ix86 Linux PC, based on GNAT 3.2 20020814. The

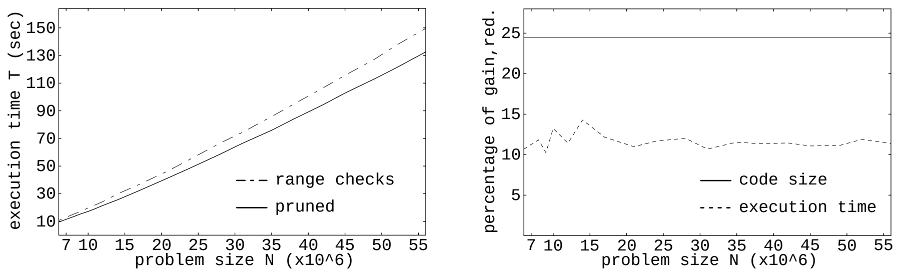

**Fig. 7.** Execution Times and Performance Gain/Obj. Code Size Reduction

left-hand picture shows absolute execution times, while the right-hand picture shows the relative performance gain and the decrease in object size.

## 6 Related Work

Determining how and when array items are accessed is of vital importance for *parallelizing compilers*. For this reason this problem has been studied extensively (see [Hag95] for a survey). Even symbolic approaches have been employed (again compare [Hag95]). On the other hand eliminating range checks has not been studied in this domain. Nevertheless it has been shown that expressions used for array indexes are linear expressions in most cases (cf. [SLY90]).

In [RR00] some kind of symbolic analysis is employed to determine upper and lower bounds of memory regions accessed by procedures and other pieces of programs. Linear programming is used to solve the problem. This approach can also be applied to our problem but it has some severe deficiencies; for example our procedure *Swap* (Fig. 1) cannot be handled correctly. In addition, if the sign of a variable is not constant (i.e., the variable does not assume only positive or negative values), the approach presented in [RR00] cannot be applied.

## 7 Conclusion and Future Work

We have presented a new method of range check elimination based on symbolic evaluation that incorporates type information provided by the underlying programming language.

Our experiments showed that affine constraints can be handled without problems by the Omega test. As a next step we will conduct investigations on large Ada code bases in order to back the assumption that most constraints are affine in nature.

In addition, we will study validity checks which requires a delicate interprocedural analysis, and overflow checks which are very costly and for this reason turned off by GNAT in its default settings.

## References

[Ada95] ISO/IEC 8652. *Ada Reference Manual*, 1995.

[BB98] J. Blieberger and B. Burgstaller. Symbolic Reaching Definitions Analysis of Ada Programs. In *Proc. of the Ada-Europe International Conference on Reliable Software Technologies*, pages 238–250, Uppsala, Sweden, June 1998.

[BBS99] J. Blieberger, B. Burgstaller, and B. Scholz. Interprocedural Symbolic Evaluation of Ada Programs with Aliases. In *Proc. of the Ada-Europe International Conference on Reliable Software Technologies*, pages 136–145, Santander, Spain, June 1999.

[BFS00] J. Blieberger, T. Fahringer, and B. Scholz. Symbolic Cache Analysis for Real-Time Systems. *Real-Time Systems, Special Issue on Worst-Case Execution Time Analysis*, 18(2/3):181–215, 2000.

[Bli02] J. Blieberger. Data-Flow Frameworks for Worst-Case Execution Time Analysis. *Real-Time Systems*, 22(3):183–227, May 2002.

[CHT79] T. E. Cheatham, G. H. Holloway, and J. A. Townley. Symbolic Evaluation and the Analysis of Programs. *IEEE Trans. on Software Engineering*, 5(4):403–417, July 1979.

[FS97] T. Fahringer and B. Scholz. Symbolic Evaluation for Parallelizing Compilers. In *Proc. of the ACM International Conference on Supercomputing*, July 1997.

[Hag95] M. R. Haghighat. *Symbolic Analysis for Parallelizing Compilers.* Kluwer Academic, 1995.

[Mae90] R. Maeder. *Programming in Mathematica.* Addison-Wesley, Reading, MA, USA, 1990.

[MP94] V. Maslov and W. Pugh. Simplifying Polynomial Constraints Over Integers to Make Dependence Analysis More Precise. In *Proc. of the International Conference on Parallel and Vector Processing*, pages 737–748, Linz, Austria, 1994.

[Pug92] W. Pugh. The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

[Rog87] H. Rogers Jr. *Theory of Recursive Functions and Effective Computability.* MIT Press, Cambridge, MA, 1987.

[RP86] B. G. Ryder and M. C. Paull. Elimination Algorithms for Data Flow Analysis. *ACM Computing Surveys (CSUR)*, 18(3):277–316, 1986.

[RR00] R. Rugina and M. Rinard. Symbolic Bounds Analysis of Pointers, Array Indices, and Accessed Memory Regions. In *Proc. of PLDI*, pages 182–195, 2000.

[SB94] E. Schonberg and B. Banner. The GNAT Project: A GNU-Ada 9X Compiler. In *Proc. of the Conference on TRI-Ada '94*, pages 48–57. ACM Press, 1994.

[SBF00] B. Scholz, J. Blieberger, and T. Fahringer. Symbolic Pointer Analysis for Detecting Memory Leaks. In *ACM SIGPLAN Workshop on "Partial Evaluation and Semantics-Based Program Manipulation"*, Boston, January 2000.

[Sed88] R. Sedgewick. *Algorithms.* Addison-Wesley, Reading, MA, USA, 2$^{nd}$ edition, 1988.

[Sho79] R. E. Shostak. A Practical Decision Procedure for Arithmetic with Function Symbols. *Journal of the ACM*, 26(2):351–360, April 1979.

[SLY90] Z. Shen, Z. Li, and P.-C. Yew. An Empirical Study of Fortran Programs for Parallelizing Compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):356–364, July 1990.

[Sre95] V. C. Sreedhar. *Efficient Program Analysis Using DJ Graphs.* PhD thesis, School of Computer Science, McGill University, Montréal, Québec, Canada, 1995.