A Formally Verified Generic Branching Algorithm for Global Optimization

Anthony Narkawicz and César Muñoz

NASA Langley Research Center, Hampton VA 23681, USA
{anthony.narkawicz,cesar.a.munoz}@nasa.gov

Abstract. This paper presents a formalization in higher-order logic of a generic algorithm that is used in automated strategies for solving global optimization problems. It is a generalization of numerical branch and bound algorithms that compute the minimum of a function on a given domain by recursively dividing the domain and computing estimates for the range of the function on each sub-domain. The correctness statement of the algorithm has been proved in the Prototype Verification System (PVS) theorem prover. This algorithm can be instantiated with specific functions for performing particular global optimization methods. The correctness of the instantiated algorithms is guaranteed by simple properties that need to be verified on the specific input functions. The use of the generic algorithm is illustrated with an instantiation that yields an automated strategy in PVS for estimating the maximum and minimum values of real-valued functions.

1 Introduction

Formal verification of safety-critical cyber-physical systems often requires proving formulas involving multivariate polynomials and other real-valued functions. For example, the following function appears in the formal proof of correctness of an alerting algorithm for parallel landing [9] in the Prototype Verification System (PVS) [12].

$$\psi(v, \phi) \equiv \frac{180\, g}{\pi v\, 0.514} \tan(\frac{\pi\phi}{180}), \tag{1}$$

where $g = 9.8$ (gravitational acceleration in meters per second squared). This formula computes the turn rate (in degrees per second) of an aircraft flying at a ground speed v (in knots) with a bank angle ϕ (in degrees). In [9], propositions involving ψ, e.g., $3 \leq \psi(250, 35) \leq 3.1$, were first checked using computer algebra tools. The mechanical, but non-automated, proof in PVS of the statement $3 \leq \psi(250, 35) \leq 3.1$ is about one page long and requires the use of several trigonometric properties.

Problems involving nonlinear real-valued functions also appear in the safety analysis of control systems. For instance, the safe domain $\mathcal{S}$ for a certain control system described in [2] is defined as follows.

E. Cohen and A. Rybalchenko (Eds.): VSTTE 2013, LNCS 8164, pp. 326–343, 2014.

$$\mathcal{S} \equiv \{(x, y) \in \mathbb{R}^2 \mid g_1(x, y) < 0 \text{ and } g_2(x, y) < 0\}, \text{where} \tag{2}$$

$$g_1(x, y) \equiv x^2y^4 + x^4y^2 - 3x^2y^2 - xy + \frac{x^6 + y^6}{200} - \frac{7}{100}, \tag{3}$$

$$g_2(x, y) \equiv -x^2y^4 - x^4y^2 + 3x^3y^3 + \frac{x^5y^3}{10} - \frac{9}{10}. \tag{4}$$

If $\mathcal{S}$ is the safe domain of a safety-critical system, the formal verification of such a system may require deciding whether or not a set of points of interest $\mathcal{I}$ is included in $\mathcal{S}$. Analytical formulas that decide these kinds of inclusions do not exist in general.

Formal modeling of biological systems can also yield non-trivial problems involving real-valued functions. Consider the polynomial

$$\begin{aligned} H \equiv -x_1x_6^3 + 3x_1x_6x_7^2 - x_3x_7^3 + 3x_3x_7x_6^2 - x_2x_5^3 + 3x_2x_5x_8^2 - x_4x_8^3 + \\ 3x_4x_8x_5^2 - 0.9563453, \end{aligned} \tag{5}$$

where $x_1 \in [-0.1, 0.4]$, $x_2 \in [0.4, 1]$, $x_3 \in [-0.7, -0.4]$, $x_4 \in [-0.7, 0.4]$, $x_5 \in [0.1, 0.2]$, $x_6 \in [-0.1, 0.2]$, $x_7 \in [-0.3, 1.1]$, and $x_8 \in [-1.1, -0.3]$. This multivariate polynomial appears in the electrolytic determination of the resultant dipole moment in the heart. Finding an enclosure to the minimum value of the polynomial in the variables' range is a challenge problem for global optimization methods [13].

The problems above, which are challenging for formal verification tools, can be solved using numerical global optimization methods [11]. One of these methods is called *branch and bound*. Branch and bound is a method to compute an enclosure to the range of a real-valued function on a given domain by recursively computing enclosures to the range of the function on subdomains. The method requires a bounding function that returns a crude, but correct, enclosure of the range of the real-valued function on any domain. The bounding function has the property of providing more accurate enclosures on smaller domains. Hence, the range of a function can be approximated to any given accuracy by splitting the original domain into two subdomains and recursively computing enclosures of the range of the function on each subdomain. This recursion continues until an appropriate enclosure is determined or until a maximum recursion depth is reached. In general, when a given domain is subdivided into two subdomains, it is possible that one of those subdomains will need further subdivision, while the other subdomain will not. That is, the recursion tree in a branch and bound algorithm is not, in general, symmetric.

Usually, in branch and bound problems, the domain of a real-valued function on n-variables is an n-dimensional hyper-rectangle, called a *box*. A box is represented as a list of closed intervals, where each interval is the range of an input variable of the function. Figure 1, which shows one possible recursion tree for a problem solved with a branch and bound algorithm. In this case, the recursion first splits the large box into left/right halves, then splits the left subbox

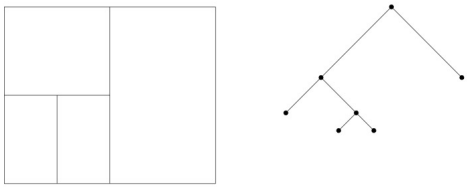

Fig. 1. Branch and Bound Recursion on a Box

into top/bottom halves, and finally splits the bottom half of this subbox into left/right halves again.

The fact that a branch and bound algorithm requires a bounding function to compute a crude estimate of a function's range on a box does not hinder the usefulness of the approach, since there are multiple ways to define such a function. One way to compute such an estimate, which works for a large class of functions, is known as *interval arithmetic* [3,7,8], which in many cases provides a worst case, naive estimate of the range of the given function. For instance, for $x \in [0,1]$, it is easy to see that the function $f(x) = x - x^2$ always takes values in $[-1,1]$, since each of the two monomials in this polynomial takes values in $[0,1]$. This estimate can be mechanically computed using interval arithmetic. This is clearly a very crude estimate of the range, since the actual range of f in $[-1,1]$ is $[0,\frac{1}{4}]$. Interval arithmetic extends to multiple variables and from polynomials to trigonometric functions, logarithms, etc. using, for example, Taylor series approximations. PVS strategies for solving simply quantified inequalities, such as those involving function ψ given by Formula (1), are presented in [3]. Those strategies are based on a branch and bound algorithm using interval arithmetic.

For polynomial functions a more accurate estimation method is available through Bernstein polynomials [5]. Any polynomial $p(x)$ of degree at most n can be written in the form of a Bernstein polynomial: $p(x) = \sum_{i=0}^{n} b_i \binom{n}{i} x^i (1-x)^{n-i}$. The coefficients b_i are called *Bernstein coefficients* and are computed directly from the coefficients of p in the power basis. Once a polynomial is written in a Bernstein polynomial form, the Bernstein coefficients yield an estimate for the range of the polynomial p for $x \in [0,1]$: $\min_{i \leq n} b_i \leq p(x) \leq \max_{i \leq n} b_i$. Furthermore, $p(0) = b_0$ and $p(1) = b_n$. This result is generally applicable to variables in an arbitrary range $[A,B]$ since any polynomial p can be translated into another polynomial q such that $q(y)$, with $y \in [0,1]$, attains the same values as $p(x)$, with $x \in [A,B]$. The case where the range of x is unbounded has been discussed in [10].

There are many types of problems that can be approached using branch and bound algorithms. Lower and upper bounds of a function on a box that are accurate to a given precision can often be computed in this way. This can be accomplished through interval arithmetic or, if the function is a polynomial,

by using Bernstein polynomials. Another problem that can be solved using a branch and bound algorithm is determining whether a given polynomial is always positive, negative, nonnegative, or nonpositive on a box. In a previous work [10], the authors presented an automated solution to this problem. That tool can be used to *automatically* and *formally* prove polynomial inequalities such as $H \geq -1.7435$, where H is the multivariate polynomial given by Formula (5). The tool itself is a collection of strategies that are implemented in PVS and are based on a branch and bound algorithm for Bernstein Polynomials [5].

Another problem that can be solved with a branch and bound algorithm is the problem of solving Boolean expressions involving more than one polynomial. This problem seems to be more common in engineering than problems with only a single polynomial. For instance, to ensure that the simply connected disk of radius 0.4 around the origin is contained in the set $\mathcal{S}$ given by Formula (2), it suffices to prove that for all $x, y \in [-1, 1]$,

$$x^2 + y^2 < 0.4^2 \text{ implies } (g_1(x, y) < 0 \text{ and } g_2(x, y) < 0).$$

This problem can be solved using a branch and bound method. A branch and bound approach can also be used to prove that the disk of radius 0.41 around the origin is not contained in $\mathcal{S}$ and to find counterexamples such as

$$(x, y) \equiv (-\frac{89186267828861}{281474976710656}, \frac{146479537812029}{562949953421312}).$$

Another global optimization problem that can be solved with a branch and bound algorithm is the problem of computing an approximation, by a list of boxes, to a set defined by a Boolean expression of polynomial inequalities. Given such a Boolean expression, three sets of subboxes of the domain can be computed: those where the property holds, those where it does not hold, and "unknown" boxes where the algorithm terminated before deciding on the truth of the expression. This problem is known as *paving*. Figure 2 shows a paving computed for the region $\mathcal{S}$, where $|x| \leq 2$ and $|y| \leq 2$, using a branch and bound algorithm [2]. The union of the green rectangles is an under-approximation of $\mathcal{S}$, the red rectangles are not in $\mathcal{S}$, and the union of the green and white rectangles is an over-approximation of $\mathcal{S}$.

This paper presents a formalization of a generic branch and bound algorithm in the higher order logic of PVS. In contrast to the branch and bound algorithms in [2, 3, 10], which use specific bounding methods and solve specific type of problems, the algorithm presented in this paper is generic with respect to the bounding function, the type of input problems, and the type of output computed by the algorithm. Since the correctness of the algorithm is formally verified in PVS, it can be used to produce strategies for automatically and formally solving a variety of global optimization problems. The use of the generic algorithm is illustrated with an instantiation that yields an automated strategy for computing estimates of the minimum and maximum value of real-valued function via interval arithmetic. The formal development presented in this paper is electronically available as part of the NASA PVS Library at `http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library`.

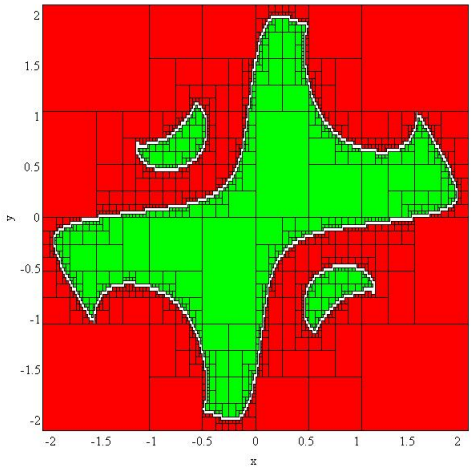

Fig. 2. Paving of a Polynomial Region

2 Generic Branch and Bound Algorithm

The motivating principle of a branch and bound algorithm is that some properties of a given function are easier to decide on small sets than on large sets. Thus, by breaking up a large domain into smaller subdomains, one can often determine whether a property holds, whereas it can not be decided easily by simply considering the large domain alone.

The rest of this section is organized as follows. The generic types used by the algorithm are described in Section 2.1. The inputs of the algorithm are given in Section 2.2. A complete description of the algorithm is presented in Section 2.3.

2.1 Generic Types

The generic branch and bound algorithm presented here depends on four generic types:

- `ObjType`: A type consisting of objects to analyze, such as Bernstein polynomials, functions admitting interval arithmetic operations, Boolean expressions of polynomial inequalities, etc.
- `AnsType`: The intended output type of the branch and bound algorithm, such as intervals containing the minimum and maximum values of a function, a Boolean value representing whether an inequality holds, a list of boxes describing an approximation to a polynomial-defined set, etc.
- `DomainType`: A type specifying exactly how (or where) elements of `ObjType` should be analyzed, such as the domain where a polynomial inequality is to be determined, or where a function is to be minimized. This may contain

not just the bounds of the interval where the polynomial is defined, but also whether it is open or closed at the boundary.
- `VarType`: A type representing variables of the objects to be analyzed. For instance, in the case of polynomials, `VarType` is a type representing the polynomial variables such as $\mathbb{N}$, where 0, 1, 2 might correspond to x_0, x_1, x_2, etc.

Specific types to substitute for these generic types are provided by the user, who chooses them with a specific application in mind. While `AnsType` is the intended output type of the branch and bound algorithm, this type is wrapped up in a larger record type that is called `Output`, which has one field consisting of an element from `AnsType`, and three other fields that give information about the execution of the function.

$$\texttt{Output} \equiv \texttt{ans} : \texttt{AnsType} \times \texttt{exit} : \texttt{boolean} \times \texttt{depth} : \mathbb{N} \times \texttt{splits} : \mathbb{N}.$$

Upon exit, the `exit` field will be set to `true` if the algorithm is forced to globally exit from the recursion, e.g., when the recursion reaches a maximum depth or a maximum number of subdivisions. As explained later, the precise conditions under which the recursion globally exits are specified by the user. The `depth` field counts the maximum recursion depth that the algorithm reaches along any branch during its execution. The `splits` field counts the total number of times that the algorithm subdivides a larger problem into two smaller problems, such as splitting a large box into two subboxes. The function `mk_out` takes as parameters an element of type `AnsType`, a Boolean value, and two naturals numbers, and builds a record of type `Output`.

The generic algorithm itself has functions for inputs, some of which depend on elements of the tuple type $\texttt{DirVar} \equiv \texttt{boolean} \times \texttt{VarType}$. During the recursion of a branch and bound algorithm, the domain often must be split in two. When this happens, a variable and a direction, i.e., `VarType` and a direction represented by a Boolean value, respectively, are selected for splitting the domain. The two halves of the original domain can be referred to by the two pairs $(\texttt{true}, j)$ and $(\texttt{false}, j)$, where j is the element of `VarType` referring to the variable chosen for subdivision. In the type `DirVar`, `true` refers to the left subdivision and `false` refers to the right subdivision.

Another type that is important for the execution of the generic algorithm is the type $\texttt{DirVarStack} \equiv \texttt{stack}[\texttt{DirVar}]$, which represents a stack of elements of type `DirVar`. The branch and bound algorithm presented in this paper implements a depth-first recursion approach. An object called `dirvars` of type `DirVar` is maintained by the algoritm, and it reflects the sequence of subdivisions, i.e., variables and directions, at any moment during the recursion.

2.2 Inputs to the Algorithm

The inputs to the generic branch and bound algorithm are listed in the table below, each with its type next to it. The element `obj` of type `ObjType` is the

Table 1. Inputs to `branch_and_bound`

Input	Type
`simplify`	[`ObjType` → `ObjType`]
`evaluate`	[`DomainType`, `ObjType` → `AnsType`]
`branch`	[`VarType`, `ObjType` → [`ObjType`, `ObjType`]]
`subdivide`	[`VarType`, `DomainType` → [`DomainType`, `DomainType`]]
`denorm`	[[`boolean`, `VarType`], `AnsType` → `AnsType`]
`combine`	[`VarType`, `AnsType`, `AnsType` → `AnsType`]
`prune`	[`DirVarStack`, `AnsType`, `AnsType` → `boolean`]
`lex`	[`AnsType` → `boolean`]
`gex`	[`DirVarStack`, `AnsType`, `AnsType` → `boolean`]
`select`	[`DirVarStack`, `AnsType`, `DomainType`, `ObjType` → [`boolean`, `VarType`]]
`accumulate`	[`AnsType`, `AnsType` → `AnsType`]
`maxd`	$\mathbb{N}$
`obj`	`ObjType`
`dom`	`DomainType`
`acc`	$Maybe$[`AnsType`]
`dirvars`	`DirVarStack`

concrete expression, e.g., a polynomial or interval expression, that the algorithm is analyzing. The element `dom` is the specific element of `DomainType` with regard to which information about `obj` is to be calculated. For example, if `ObjType` consists of polynomials, `DomainType` may consist of boxes that constraint the range of the polynomial variables.

The function `simplify` rewrites `obj` to make it easier to manipulate before carrying out the next recursive calculations. In many cases this function is just the identity function. However, in some cases, a canonical form of `obj` may yield more efficient computations.

The function `evaluate` gives a crude estimate of an element of `AnsType` that would describe `obj` for a particular element of `DomainType`. For instance, if `obj` is a function and the algorithm is being used to give a precise estimate for the range of the function over a box, `evaluate` may use interval arithmetic to give a crude estimate of the range on a specific domain without splitting this domain in two.

The function `branch`, for a specific element of `VarType`, takes an element of `ObjType` and gives two more elements of `ObjType` that correspond to splitting the problem in two. For instance, if `VarType` corresponds to the possible variables of a function, and if `ObjType` consists of polynomials written in Bernstein form, then given a specific variable x_j, the function `branch` will take a polynomial and turn it into two polynomials. Each of these polynomials represents the original polynomial on half of the original unit box, which has been split along the variable x_j. The two polynomials themselves are translated from each of these half boxes back to the unit box by changing the variable x_j linearly. These polynomials each represent the original polynomial on half of the original box after a linear translation in x_j.

As explained before, the input `dirvars` reflects the subdivision that have occurred up to the current recursive call. The value of this parameter at every recursive step is maintained by the algorithm. The initial value, provided by the user, is expected to be an empty stack. The value of `dirvars` can be used by several other input functions to choose the variable for subdivision at the current step, determine whether to prune the recursion tree, or decide to exit globally from the recursion.

The function `subdivide` takes an element of `DomainType` and divides it into two new elements of `DomainType`, where the exact division is specified by an element of `VarType`. For instance, `DomainType` may consist of boxes and `VarType` to variables, in which case `subdivide` may split a box in the middle along a given variable.

The function `denorm` translates an element of `AnsType`, which gives information about the object on one half of an element of `DomainType`, after that element has been split using `subdivide`, back to level of the original, non-subdivided object in the recursion. For instance, if the algorithm is designed to find a counterexample to a polynomial inequality and a box is split into two subboxes, then a counterexample found by the algorithm on one of these subboxes is also a counterexample on the larger box. In this example, the function `denorm` would translate the point where the counterexample was found to a point in the original box. The first parameter of this function, which has the type `VarType`, would represent the variable along which the original box was split, and the Boolean value would represent whether this particular subbox was the right half or the left half of the original.

The function `combine` takes two elements of type `AnsType`, each giving information about the object in question on half of the original box, and combines them into one element of `AnsType` that gives information about the object on the larger box. It depends on an element of type `VarType`, which may, for example, represent the variable along which the original box was split to give the two subboxes in question.

The function `prune` decides whether to locally exit the recursion at the current step and continue the recursion at the next step without subdividing the problem further on the current branch. It takes an accumulated value of type `AnsType` from previous steps in the recursion, namely the element `acc`, which gives information about the object in question at locations other than the current location in the domain, and uses this to decide whether it is beneficial to continue down the current branch in the recursion. For example, consider an algorithm finding an interval that is guaranteed to contain the minimum value attained by the a function. Suppose that in a branch of the recursion, this interval was reduced to $[0.9, 1.0]$, which will be stored in the `acc` element of `AnsType`. If on the current branch, the `evaluate` function gives $[1.2, 1.3]$ as as a crude estimate of the minimum on a small subset of the larger domain, then the recursion can often be stopped from further continuing down the current branch, since the minimum will not be found on the current branch.

The accumulated value `acc` that is passed as a parameter to `prune` is computed by the function `accumulate`. This function combines all of the information from previous recursive steps, along with the information gained at the current step, in the element `acc` for use further down the current branch and in other branches still unexplored. Depending on the problem that the branch and bound algorithm is trying to solve, there are cases where previous information cannot be reused in a different branch. This possibility is handled by the return type `Maybe[AnsType]` of the function `accumulate`. This type represents an undefined value, represented by `None`, or an actual value v of type `AnsType`, represented by `Some`(v). The functions `none?` and `some?` check if an element of type `Maybe` is either `None` or `Some`(v) for some v, respectively. In the latter case, the value v can be accessed with the function `val`.

The `lex` function, which stands for *local exit*, determines when the function has locally succeeded and therefore does not need to subdivide on the current branch anymore. It considers an element of type `AnsType` given by the output of `evaluate`, and uses this information to determine success. For example, if the algorithm is proving that a function is always nonnegative, and if the function `evaluate` indicates that it is true on the small subbox represented at this point in the recursion tree, then the algorithm has proved the result on this local subbox and does not need to divide the subbox further. It then moves on with the recursion elsewhere.

The function `gex`, which stands for *global exit*, determines whether, at the current recursion step, the algorithm should exit completely from the recursion without computing anything else. This is desirable when the recursion has reached a depth that is larger than the user wants, and it is also desirable when the algorithm has found a satisfactory answer at the current recursive step and no longer needs to continue the recursion. One example of a condition that warrants a global exit is when the algorithm is searching for a counterexample to the positivity of a function and it finds such a counterexample at the current step.

The function `select` determines where the next subdivision will occur. For example, if `DomainType` consists of boxes in n variables, then any subdivision of a box will occur along a particular variable. In this context, the `select` function will determine the variable along which to subdivide and a Boolean value representing whether the recursion should first compute either the left subdivision or the right one. An additional parameter to this function is `dirvars`, which gives information about the other variables that have been chosen for subdivision at previous steps in the current branch. This allows `select` to be defined in a way that is fair, meaning that in every possible infinite branch of the infinite recursion tree, every variable occurs an infinite number of times.

The input `maxd` represents a maximum recursion depth and removes the possibility of a non-terminating algorithm. When the current depth reaches `maxd`, the algorithm forces a local exit as opposed to a global one. Depending on the problem that the branch and bound algorithm is solving, an output can still be sound even if some branches have reached the maximum depth. Hence, even

though the algorithm always terminates, it may take a very long time to do so. If the user wants to specify a global exit when the maximum depth `maxd` is reached, this has to be done through the input function `gex`.

2.3 The Branching Algorithm

The algorithm `branch_and_bound` is defined in Figure 3. Lines 1-12 define the bounding and pruning aspects of the branch and bound algorithm. These lines concern the computation of a crude estimate for the object `obj` in the domain `dom` (Line 5). It is noted that the estimate, namely `thisans`, is actually computed for the object `thisobj`, which is intended to be a simplified version of `obj` (Line 4). The accumulated value `thisacc` is computed from the previous value `acc` and the computed answer `thisans` (Line 6). The function `gex` uses the information on `thisacc`, `thisans`, and the stack `dirvars` to determine if the algorithm should stop (Line 8). This information is propagated to the remaining recursive calls through the field `exit` in `thisans`. If the answer value `thisans` is good enough, which is determined by the functions `lex` and `prune`, or if the maximum depth `maxd` is reached, the current recursive call ends (Lines 11-12). In this case, the output consists of the answer `thisans`, the Boolean value computed by the function `gex`, a value of depth that is equivalent to the length of the stack `dirvars`, and 0, which represents the number of splits. The function `mk_out` builds such a record of type `Output` (Line 9).

The branching aspect of the algorithm is defined in Lines 14-39. First, a direction `dir` and a variable v are selected for subdivision (Line 14). Subdivided objects $\texttt{obj}_l, \texttt{obj}_r$ and subdivided domains $\texttt{dom}_l, \texttt{dom}_r$ are computed accordingly (Line 15-18). Then, the first recursive call, in the direction determined by `dir`, is made (Lines 19-21). When the recursion returns, it may be the case that a global exit was signaled during the previous recursive call (Line 24). In this case, an answer is computed from the value returned by the recursive call and the current answer `thisans` (Lines 24-25). In order to combine these values, the answer from the subdivided domain $\texttt{dom}_1$ has to be translated to the whole domain `dom`. This is performed by the function `denorm` (Line 24). The values of the fields `depth` and `splits` in the output record are computed appropriately (Line 25).

The second recursive call is specified in Lines 27-35. In this case, the answers returned by the two calls, for each one of the subdivisions, are combined into an answer for the whole domain (Line 32). The output record consists of this combined answer and appropriate values for the fields `exit`, `depth`, and `splits` (Lines 34-35).

3 Correctness of the Algorithm

The output of the function `branch_and_bound` in Figure 3 has type `Output`. In order for the algorithm to be useful for solving problems in global optimization, the element of `Output` returned by the algorithm must satisfy a correctness property. Not only does the algorithm `branch_and_bound` take a generic set of

```
01 : branch_and_bound(simplify, evaluate, branch, subdivide, denorm, combine, prune,
02 :         lex, gex, select, accumulate, maxd, obj, dom, acc, dirvars) : Output ≡
03 :   let
04 :     thisobj = simplify(obj),
05 :     thisans = evaluate(dom, thisobj),
06 :     thisacc = if none?(acc) then thisans
07 :                 else accumulate(val(acc), thisans) endif,
08 :     thisexit = gex(dirvars, thisacc, thisans),
09 :     thisout = mk_out(thisans, thisexit, length(dirvars), 0)
10 :   in
11 :     if length(dirvars) = maxd or lex(thisans) or thisexit or
12 :        prune(dirvars, thisacc, thisans) then thisout
13 :     else let
14 :          (dir, v) = select(dirvars, thisacc, dom, thisobj),
15 :          (obj_l, obj_r) = branch(v, thisobj),
16 :          (obj_1, obj_2) = if dir then (obj_l, obj_r) else (obj_r, obj_l) endif,
17 :          (dom_l, dom_r) = subdivide(v, dom),
18 :          (dom_1, dom_2) = if dir then (dom_l, dom_r) else (dom_r, dom_l) endif,
19 :          out_1 = branch_and_bound(simplify, evaluate, branch, subdivide, denorm,
20 :                          combine, prune, lex, gex, select, accumulate, maxd,
21 :                          obj_1, dom_1, thisacc, push((dir, v), dirvars))
22 :        in
23 :          if out_1.exit then
24 :            mk_out(combine(v, denorm((dir, v), out_1.ans), thisans), true,
25 :                   out_1.depth, out_1.splits + 1)
26 :          else let
27 :               newacc = accumulate(thisacc, out_1.ans),
28 :               out_2 = branch_and_bound(simplify, evaluate, branch, subdivide, denorm,
29 :                               combine, lex, gex, select, accumulate, maxd,
30 :                               obj_2, dom_2, newacc, push((¬dir, v), dirvars))
31 :               (out_l, out_r) = if dir then (out_1, out_2) else (out_2, out_1) endif,
32 :               ans = combine(v, denorm((true, v), out_l.ans), denorm((false, v), out_r.ans))
33 :             in
34 :               mk_out(ans, out_2.exit, max(out_1.depth, out_2.depth),
35 :                       out_1.splits + out_2.splits + 1)
36 :          endif
37 :     endif
```

Fig. 3. The function branch_and_bound

inputs with numerous possible instantiations, but its correctness property is generic as well. This correctness property is represented by the abstract predicate *sound?*, which has the type indicated below.

$$sound?: [\texttt{DomainType}, \texttt{ObjType}, \texttt{AnsType} \to \texttt{boolean}]$$

The strength of a generic branching algorithm such as `branch_and_bound` relies on the fact that it reduces the correctness proof of a particular instantiation to proving simpler statements about the compatible behavior of the input functions `evaluate`, `simplify`, `subdivide`, `branch`, `denorm`, and `combine`. The correctness property depends *only* on these input function parameters. In particular, the generic algorithm has been proved to be *sound* for any particular instantiation of the functions `lex`, `gex`, `prune`, and `select`. Those functions are usually the most technically involved since they deal with heuristics that improve the efficiency of the algorithm. All of these concerns are abstracted away in the correctness stament of the algorithm.

The main correctness result is stated as follows.

Theorem 1. *For all inputs that satisfy*

- *accommodates?*(*sound?*, `evaluate`),
- *simplify_invariant?*(*sound?*, `simplify`),
- *evaluate_simplify?*(`evaluate`, `simplify`),
- *branch_simplify?*(`branch`, `simplify`),
- *subdiv_presound?*(*sound?*, `subdivide`, `branch`, `denorm`, `combine`), *and*
- *subdiv_sound?*(*sound?*, `subdivide`, `branch`, `denorm`, `combine`),

sound?(`dom`, `obj`, `bnb.ans`) *is* `true`, *where* `bnb` *is equal to*

$$\texttt{branch\_and\_bound}(\texttt{simplify}, \texttt{evaluate}, \texttt{branch}, \texttt{subdivide}, \texttt{denorm}, \texttt{combine}, \texttt{prune}, \texttt{lex}, \texttt{gex}, \texttt{select}, \texttt{accumulate}, \texttt{maxd}, \texttt{obj}, \texttt{dom}, \texttt{None}, \texttt{empty\_stack}).$$

The proof of this theorem, which has been mechanically verified in PVS, proceeds by induction on $\texttt{maxd} - \texttt{length}(\texttt{dirvars})$. The predicates *accommodates?*, *simplify_invariant?*, *evaluate_simplify?*, *branch_simplify?*, *subdiv_presound?* and *subdiv_sound?* are defined as follows.

The predicate *accommodates?* states that the function `evaluate` computes a valid estimate for the object `obj` on the domain `dom`.

$$accommodates?(sound?, \texttt{evaluate}) \equiv \forall\,(\texttt{dom}, \texttt{obj}) : sound?(\texttt{dom}, \texttt{obj}, \texttt{evaluate}(\texttt{dom}, \texttt{obj})).$$

The predicate *simplify_invariant?* states the function `simplify` preserves soundness.

$$simplify\_invariant?(sound?, \texttt{simplify}) \equiv \forall\,(\texttt{dom}, \texttt{obj}, \texttt{ans}) : sound?(\texttt{dom}, \texttt{obj}, \texttt{ans}) \iff sound?(\texttt{dom}, \texttt{simplify}(\texttt{obj}), \texttt{ans}).$$

The predicate *evaluate_simplify?* states that simplified objects evaluate to the same value.

$$\begin{aligned} &\mathit{evaluate\_simplify?}(\texttt{evaluate}, \texttt{simplify}) \equiv \forall\,(\texttt{dom}, \texttt{obj}) : \\ &\qquad \texttt{evaluate}(\texttt{dom}, \texttt{obj}) = \texttt{evaluate}(\texttt{dom}, \texttt{simplify}(\texttt{obj})). \end{aligned}$$

The predicate *branch_simplify?* states that the function `simplify` and `branch` commute.

$$\begin{aligned} &\mathit{branch\_simplify?}(\texttt{branch}, \texttt{simplify}) \equiv \forall (v, \texttt{obj}) : \\ &\qquad \texttt{let } (\texttt{obj}_l, \texttt{obj}_r) = \texttt{branch}(v, \texttt{obj}) \texttt{ in} \\ &\qquad\quad \texttt{branch}(v, \texttt{simplify}(\texttt{obj})) = (\texttt{simplify}(\texttt{obj}_l), \texttt{simplify}(\texttt{obj}_r)). \end{aligned}$$

The last two predicates specify the core behavior of the functions `subdivide`, `branch`, `denorm`, and `combine`. They express that the soundness of the output on the whole domain can be deduced from the soundness of the outputs on the subdivided domains. The former predicate refers to the case where only one branch is recursively explored. The latter predicate refers to the case where both left and right branches are recursively explored.

$$\begin{aligned} &\mathit{subdiv\_presound?}(\mathit{sound?}, \texttt{subdivide}, \texttt{branch}, \texttt{denorm}, \texttt{combine}) \equiv \\ &\qquad \forall\,(v, \texttt{dom}, \texttt{obj}, \texttt{dir}, \texttt{ans}_1, \texttt{ans}_2) : \\ &\qquad\quad \texttt{let } (\texttt{dom}_l, \texttt{dom}_r) = \texttt{subdivide}(v, \texttt{dom}), \\ &\qquad\qquad\quad (\texttt{obj}_l, \texttt{obj}_r) = \texttt{branch}(v, \texttt{obj}) \\ &\qquad\quad \texttt{in } \mathit{sound?}(\texttt{dom}, \texttt{obj}, \texttt{ans}_1) \text{ and} \\ &\qquad\qquad (\texttt{dir} \implies \mathit{sound?}(\texttt{dom}_l, \texttt{obj}_l, \texttt{ans}_2)) \text{ and} \\ &\qquad\qquad (\neg\texttt{dir} \implies \mathit{sound?}(\texttt{dom}_r, \texttt{obj}_r, \texttt{ans}_2)) \\ &\qquad\quad \implies \mathit{sound?}(\texttt{dom}, \texttt{obj}, \texttt{combine}(v, \texttt{denorm}((\texttt{dir}, v), \texttt{ans}_2, \texttt{ans}_1)). \end{aligned}$$

$$\begin{aligned} &\mathit{subdiv\_sound?}(\mathit{sound?}, \texttt{subdivide}, \texttt{branch}, \texttt{denorm}, \texttt{combine}) \equiv \\ &\qquad \forall\,(v, \texttt{dom}, \texttt{obj}, \texttt{dir}, \texttt{ans}_1, \texttt{ans}_2) : \\ &\qquad\quad \texttt{let } (\texttt{dom}_l, \texttt{dom}_r) = \texttt{subdivide}(v, \texttt{dom}), \\ &\qquad\qquad\quad (\texttt{obj}_l, \texttt{obj}_r) = \texttt{branch}(v, \texttt{obj}) \\ &\qquad\quad \texttt{in } \mathit{sound?}(\texttt{dom}_l, \texttt{obj}_l, \texttt{ans}_1) \text{ and } \mathit{sound?}(\texttt{dom}_r, \texttt{obj}_r, \texttt{ans}_2) \\ &\qquad\quad \implies \mathit{sound?}(\texttt{dom}, \texttt{obj}, \texttt{combine}(v, \texttt{denorm}((\texttt{true}, v), \texttt{ans}_1), \\ &\qquad\qquad\qquad\qquad\qquad\qquad \texttt{denorm}((\texttt{false}, v), \texttt{ans}_2))). \end{aligned}$$

Theorem 1 is significantly simpler when the function `simplify` is the identity. The next corollary considers this case.

Corollary 1. *Let I be the identity function on the type `ObjType`. For all inputs that satisfy*

– *accommodates?*(*sound?*, `evaluate`),
– *subdiv_presound?*(*sound?*, `subdivide`, `branch`, `denorm`, `combine`), *and*
– *subdiv_sound?*(*sound?*, `subdivide`, `branch`, `denorm`, `combine`),

sound?(`dom`, `obj`, `bnb.ans`) *is* `true`, *where* `bnb` *is equal to*

$$\texttt{branch\_and\_bound}(I, \texttt{evaluate}, \texttt{branch}, \texttt{subdivide}, \texttt{denorm}, \texttt{combine}, \texttt{prune}, \texttt{lex}, \texttt{gex}, \texttt{select}, \texttt{accumulate}, \texttt{maxd}, \texttt{obj}, \texttt{dom}, \texttt{None}, \texttt{empty\_stack}).$$

4 Branch and Bound Algorithm for Interval Expressions

This section presents an instantiation of the function `branch_and_bound` in Figure 3 that yields a strategy in PVS for computing estimates of the minimum and maximum values of a multivariate real-valued functions. These estimates are found using interval arithmetic.

In order to define this instantiation, it is necessary to provide a deep embedding of arithmetic expressions. Such an embedding has been developed and is available as part of the interval arithmetic development in the NASA PVS Library.[1] The abstract data type `IntervalExpr`, which is part of the library, represents arithmetic expressions constructed from basic operations, power, absolute value, square root, trigonometric functions, the irrational constants π and e, the exponential and logarithm functions, numerical constants, and variables that range over closed intervals. Henceforth, elements of type `IntervalExpr` are called *expressions*. The following types and functions are also available.

– `Interval`: A tuple of two elements that represents the upper and lower bounds of a closed, non-empty, interval. Elements of this type are called *intervals*.
– `Box`: A list of elements of type `Interval`. Elements of this type are called *boxes*.
– `Env`: A list of real numbers representing an evaluation environment for the variables in a given expression. Elements of this type are called *environments*.
– `well_typed?`: A predicate that has as parameters a box B and an expression E. The predicate holds when E is well-defined in B. This predicate is used to avoid the case of division by zero.
– `eval`: A function that has as parameters an expression E and an environment Γ. The function returns a real value that corresponds to the evaluation of E in Γ.
– `Eval`: A function that has as parameters an expression E and a box B. The function returns an interval value that correspond to the interval arithmetic evaluation of E in B.

The following two key theorems of interval arithmetic are mechanically proved in PVS.

[1] `http://shemesh.larc.nasa.gov/people/cam/Interval`

Theorem 2 (Inclusion Theorem). *For all B, E, and Γ,*

$$\mathit{well\_typed?}(B, E) \text{ and } \Gamma \in B \text{ implies } \mathit{eval}(E, \Gamma) \in \mathit{Eval}(E, B).$$

Theorem 3 (Fundamental Theorem). *For all $B_1 \subseteq B_2$ and E,*

$$\mathit{well\_typed?}(B_2, E) \text{ implies } \mathit{Eval}(E, B_1) \subseteq \mathit{Eval}(E, B_2).$$

A simple of instantiation of the generic branch and bound algorithm is obtained as follows. The parameter types `ObjType`, `AnsType`, and `DomainType` are instantiated with the concrete types `IntervalExpr`, `Interval`, and `Box`. The parameter type `VarType`, representing variables in `IntervalExpr`, is instantiated with the concrete type `nat`. Furthermore,

- the function `evaluate` is defined as $\texttt{evaluate}(B, E) \equiv \texttt{Eval}(E, B)$,
- the function `branch` is defined as $\texttt{branch}(n, E) \equiv (E, E)$,
- the function `subdivide` is defined as $\texttt{subdivide}(n, B) \equiv \texttt{split}(n, B)$ that returns two boxes that are equal to B except in their n-th interval, where the original interval is divided into mid-left and mid-right intervals,
- the functions `combine` and `accumulate` are both defined as the union of two intervals,
- the functions `simplify` and `denorm` are defined as identity functions on the types `IntervalExpr` and `Interval`, respectively.

Since the soundness theorem of the generic branch and bound algorithm does not depend on the predicates `gex`, `lex`, or `prune`, they can be arbitrarily instantiated. In particular, they are instantiated such that they always return `false`. This means that the instantiated branch and bound algorithm, called `simple_interval`, completely explores the recursion tree up to the maximum depth. The direction and variable selection function `select` is simply defined using a round-robin approach.

$$\texttt{simple\_interval}(\texttt{maxd}, E, B) : \texttt{Output} \equiv$$
$$\texttt{branch\_and\_bound}(\texttt{simplify}, \texttt{evaluate}, \texttt{branch}, \texttt{subdivide}, \texttt{denorm},$$
$$\texttt{combine}, \texttt{prune}, \texttt{lex}, \texttt{gex}, \texttt{select}, \texttt{accumulate}, \texttt{maxd}, E, B).$$

The intended soundness property of the function `simple_interval` is expressed by the following predicate.

$$\mathit{sound?}(B, E, \texttt{ans}) \equiv \texttt{well\_typed?}(B, E) \Longrightarrow \forall(\Gamma \in B) : \texttt{eval}(E, \Gamma) \in \texttt{ans}.$$

The following theorem has been proved in PVS. It follows from Corollary 1. The fact that the predicate *accommodates?* holds follows directly from the Inclusion Theorem (Theorem 2). The properties concerning *subdiv_presound?* and *subdiv_sound?* are consequences of the Fundamental Theorem (Theorem 3).

Theorem 4. *For any maximum depth* `maxd`*, expression E, and box B,* $\mathit{sound?}(B, E, \mathit{simple\_interval}(\mathit{maxd}, E, B).\mathit{ans})$ *holds.*

The function `simple_interval` and its correctness property (Theorem 4) are the basis of a computational reflection strategy in PVS for computing estimates of the minimum and maximum values of real expressions. The strategy, called `simple-numerical`, takes a PVS real expression, possibly involving variables, and reflects it in the type `IntervalExpr`. The key step in the strategy is the ground evaluation of the function `simple_interval`, which implements the instantiated branch and bound algorithm. Theorem 4 guarantees the soundness of the computation in the PVS logic. For instance, using `simple_interval`, it can be automatically proved the statement $|\psi(v, \phi)| \leq 3.825$, where ψ is defined as in Formula (1), $v \in [200, 250]$, and $|\phi| \leq 35$. This result, which is used in the correctness proof of an alerting algorithm for aircraft performing a parallel landing [9], states that for an aircraft flying at a ground speed between 200 and 250 knots, the maximum angular speed is less than 4 degrees (more precisely, less than 3.825 degrees), assuming a maximum bank angle of 35 degrees.

The development `interval_arith` in the NASA PVS Library includes more sophisticated instantiations of `branch_and_bound` and strategies based on these instantiations for computing estimates of the minimum and maximum values of real-valued functions up to a precision provided by the user and for proving real-valued inequalities. These instantiations make use of the input functions `select`, e.g., for implementing better heuristics to chose the direction of the branching and the variable to subdivide, of `lex`, e.g., for stopping the current branch when a given precision is reached, of `gex`, e.g., for stopping the recursion when a given inequality cannot be proved, and of `prune`, e.g., for pruning a branch when the recursion will not improve the accumulated value.

5 Conclusion

The generic branch and bound algorithm presented in this paper has been used in several contexts, e.g., computing the range of a function on a box using interval arithmetic, computing the range of a polynomial on a box using Bernstein polynomials, deciding whether a simply quantified polynomial inequality holds on a box, deciding whether a Boolean expression involving polynomial inequalities holds on a box, and paving a region defined by polynomial inequalities. For each of these instantiations, the correctness statement follows almost immediately from the correctness statement for the generic algorithm. In each case, this requires only proving certain properties about the input functions to the generic algorithm.

Strategies similar to the one described in Section 4, based on interval arithmetic and subdivision, are available in PVS [3], Coq [6], HOL Light [14], etc. The novelty of the work presented in this paper is not the development of interval arithmetic strategies, but the fact these strategies are implemented on top of a formally verified generic branching algorithm that can be instantiated with different domains. In addition to instances related to interval arithmetic and Bernstein polynomials, many other instances of the generic branch and bound algorithm are being considered including Taylor models and affine arithmetic.

The approach presented in this paper is similar, in spirit, to that of Carlier et al. [1] on the verification of a constraint solver, where the application domain is abstracted away. However, the emphasis here is to support the development of efficient automated strategies that execute the generic algorithm via computational reflection [4]. Indeed, even the simple interval arithmetic presented here, which fully explores the recursion tree, is significantly more efficient that the strategies presented in [3]. Furthermore, since the correctness statement of the algorithm does not depend on parameter functions for variable selection method and pruning, they can be freely instantiated for implementing advanced heuristics. In the case of Bernstein polynomials, this feature has allowed the authors to experiment with different pruning heuristics for the algorithm proposed in [10].

References

1. Carlier, M., Dubois, C., Gotlieb, A.: A certified constraint solver over finite domains. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 116–131. Springer, Heidelberg (2012)
2. Crespo, L.G., Muñoz, C.A., Narkawicz, A.J., Kenny, S.P., Giesy, D.P.: Uncertainty analysis via failure domain characterization: Polynomial requirement functions. In: Proceedings of European Safety and Reliability Conference, Troyes, France (September 2011)
3. Daumas, M., Lester, D., Muñoz, C.: Verified real number calculations: A library for interval arithmetic. IEEE Transactions on Computers 58(2), 1–12 (2009)
4. Harrison, J.: Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK (1995), http://www.cl.cam.ac.uk/jrh13/papers/reflect.dvi.gz+
5. Lorentz, G.G.: Bernstein Polynomials, 2nd edn. Chelsea Publishing Company, New York (1986)
6. Melquiond, G.: Proving bounds on real-valued functions with computations. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 2–17. Springer, Heidelberg (2008)
7. Moa, B.: Interval Methods for Global Optimization. PhD thesis, University of Victoria (2007)
8. Moore, R.E., Kearfott, R.B., Cloud, M.J.: Introduction to Interval Analysis. Cambridge University Press (2009)
9. Muñoz, C., Carreño, V., Dowek, G., Butler, R.: Formal verification of conflict detection algorithms. International Journal on Software Tools for Technology Transfer 4(3), 371–380 (2003)
10. Muñoz, C., Narkawicz, A.: Formalization of a Representation of Bernstein Polynomials and Applications to Global Optimization. Journal of Automated Reasoning 51(2), 151–196 (2013), http://dx.doi.org/10.1007/s10817-012-9256-3, doi:10.1007/s10817-012-9256-3
11. Neumaier, A.: Complete search in continuous global optimization and constraint satisfaction. Acta Numerica 13, 271–369

12. Owre, S., Rushby, J., Shankar, N.: PVS: A prototype verification system. In: Kapur, D. (ed.) CADE 1992. LNCS, vol. 607, pp. 748–752. Springer, Heidelberg (1992)
13. Ray, S., Nataraj, P.S.: An efficient algorithm for range computation of polynomials using the Bernstein form. Journal of Global Optimization 45, 403–426 (2009)
14. Solovyev, A., Hales, T.C.: Formal verification of nonlinear inequalities with Taylor interval approximations. In: Brat, G., Rungta, N., Venet, A. (eds.) NFM 2013. LNCS, vol. 7871, pp. 383–397. Springer, Heidelberg (2013)