Enclosure Constraints for Floating Point Software Verification

Jan A. Duracz and Amin Farjudian and Michal Konečný

Computer Science, Aston University

Abstract. SPARK is an environment for the development and validation of software for high integrity applications. So far the SPARK annotation language and tool set have lacked explicit support for floating point arithmetic. This paper describes ongoing work extending the annotation language to support analysis and proof of functional properties and exception freedom for SPARK floating point programs.
We present an annotation language for floating point programs based on enclosure constraints with interval expressions. Then we show how to use this language to annotate an example program in SPARK. From these annotations SPARK generates verification conditions for accuracy guarantees and for absence of division by zero exceptions.
We give proofs of these verification conditions and discuss the connection our approach has to domain theory. We also address the automation of such proofs using function enclosure arithmetic. We point out some implementation challenges and suggest solutions outlining necessary modifications of an existing special purpose theorem prover.

1 Introduction

Verification of floating point software has been studied extensively, resulting in a number of practical applications, such as the verification of aeronautical software by Blanchet et al [1]. In the verification of large programs automation of simple but tedious proofs becomes essential. Automated numerical theorem provers [2, 3] based on techniques from the numerical constraint satisfaction field have been proposed [4] to tackle proofs of *verification conditions* (VCs) arising from floating point verification problems.

The work presented in this paper addresses the problem of specifying numerical properties of floating point code for the purpose of automating partial correctness proofs. The main contributions are a set of new annotation language primitives that facilitate the expression of such properties in a concise form and a theoretical analysis of the semantics of the resulting correctness theorems. This is addressed mostly in Sect. 2. Then, in Sect. 3 we illustrate the use of these annotations on a simple yet instructive program, for which we prove a numerical accuracy guarantee and absence of division by zero exceptions.

Finally, in Sect. 4 we discuss the automation of the proofs, the challenges thereof and various ways of overcoming them. In particular, we point out and elaborate on the use of *functional enclosures* as a way of speeding up the decision procedures and we argue for the use of *inconsistent enclosures* forming *interval lattices* in the proofs of some of our VCs.

2 Annotation Language with Enclosures

An annotation language is a language embedded within a programming language. Usually, annotations are entered into program code as comments, allowing the programmer to formally express program properties without changing the operational semantics of the program. In SPARK [5], the annotation language essentially consists of first order predicate logic formulae over program variables. A program variable may be of a floating point *subtype*, which is the data type consisting of floating point numbers in a range $[f_l, f_r]$. In the annotations this range is interpreted as the interval $[f_l, f_r]$ of *real numbers*. This includes the set of all finite floating point numbers, in which case the interval would be $[f_{\min}, f_{\max}]$, where $f_{\min}$ and $f_{\max}$ are the smallest and largest representable floating point numbers, respectively.

Annotation languages aimed at verification of functional properties and the absence of numerical exceptions should provide a convenient way of expressing these properties. The current SPARK annotation language lacks explicit support for floating point error analysis and the work presented in this article is an attempt to fill this gap. Currently, floating point arithmetic is treated as exact real arithmetic, ignoring rounding errors. Below, we propose an extension to the SPARK annotation language, adding primitives tailored for the purpose of floating point verification, allowing the generation of VCs for partial correctness of numeric properties. The proposed extension offers succinct expression of such properties, as demonstrated in Sect. 3. The properties — roughly speaking bounds on computed values — are compositional in the sense that intermediate variables may be eliminated automatically, resulting in VCs that depend on fewer variables. Reducing the number of variables in VCs is essential when attempting to prove them automatically.

2.1 Expressing constraints on rounding errors

On a machine with exact arithmetic, the functional properties of a computation may be given in terms of the real function that is computed. Functional properties of an inexact computation may be given in terms of suitable enclosures of the function for the corresponding exact computation [4]. The Ada Reference Manual [6] uses precisely this kind of properties to describe the numerical accuracy guarantees for elementary functions.

Accuracy guarantees given in the Numeric Annex of the Ada Reference Manual state that the floating point result r of calling a given elementary floating point function `f` with a floating point argument x lies[1] in the smallest floating point interval containing the set

$$\{(1 + d) f(x) \mid |d| \leq \varepsilon_f(x)\} \tag{1}$$

where ε_f is called the function's *maximum relative error* and $f(x)$ denotes the value of the corresponding (possibly partial) function $f : \mathbb{R} \to \mathbb{R}$ at x. On an IEEE-754

[1] To be precise, x must be inside the *safe range* of the function f, i. e. the range of values for which the floating point computation of $f(x)$ does not lead to exceptions such as overflow, division by zero, etc.

compliant [7] system, the rounding error made when fitting a real number x in a floating point format may — provided x does not round to an infinity — be bounded by

$$\max(|x|\varepsilon_{rel}, \varepsilon_{abs}) \tag{2}$$

where ε_{rel} and ε_{abs} are called the *relative error* and *absolute error* of the format. ε_{rel} is defined as the smallest positive number in the format for which the equation

$$1.0 + \varepsilon_{rel} \neq 1.0$$

holds, while ε_{abs} is defined as the smallest normalised number in the format. Below is a list of the maximum relative errors for functions in the Ada numerics package, defined in terms of ε_{rel}

- `sqrt`, `sin` and `cos` have[2] a maximum relative error of $2\varepsilon_{rel}$
- `log`, `exp`, `tan` and `cot` have a maximum relative error of $4\varepsilon_{rel}$
- the forward and inverse hyperbolic functions have a maximum relative error of $8\varepsilon_{rel}$
- the maximum relative error for exponentiation x^y depends on the parameter values x and y, it is $\left(4 + \frac{|y \log(x)|}{32}\right)\varepsilon_{rel}$

Denoting rounding by $\mathcal{R}$ we can express the above guarantees in the following form

$$\exists d \in [-\varepsilon_f, \varepsilon_f].r = \mathcal{R}((1+d)f(x)) \tag{3}$$

or equivalently, using an enclosure inclusion

$$r \in \mathcal{R}_{out}\Big((1 + [-\varepsilon_f, \varepsilon_f])f(x)\Big) \tag{4}$$

where the outward rounding $\mathcal{R}_{out}$ thickens the enclosure so that it includes all values that may result from rounding elements in the enclosure.

Compositional constraints. An alternative to using constraints such as (3) or (4) is to bound r by two expressions:

$$\begin{cases} r \leq \mathcal{R}_{+\infty}\Big(\big(1 + \text{sign}\,(f(x))\varepsilon_f(x)\big)f(x)\Big) \\ r \geq \mathcal{R}_{-\infty}\Big(\big(1 - \text{sign}\,(f(x))\varepsilon_f(x)\big)f(x)\Big) \end{cases} \tag{5}$$

The main problem with (5) is that it de-couples the bounds on r, making it difficult to mechanically reason with such constraints. As an example, consider the program in Fig. 1 on the following page, computing the composition `sin(cos(`x`))`. Formulating the accuracy guarantees for `sin` and `cos` as in (5), we obtain the information about r shown in (6).

[2] Ada trigonometric functions have an additional parameter for the period of the function. The accuracy guarantees hold whenever the argument value is taken from the period interval.

```
1 y := cos(x);
2 r := sin(y);
```

Fig. 1. Example of function composition.

$$
\begin{aligned}
y &\leq \mathcal{R}_{+\infty}\Big(\big(1 + \operatorname{sign}(\cos(x))2\varepsilon_{rel}\big)\cos(x)\Big)\\
y &\geq \mathcal{R}_{-\infty}\Big(\big(1 - \operatorname{sign}(\cos(x))2\varepsilon_{rel}\big)\cos(x)\Big)\\
r &\leq \mathcal{R}_{+\infty}\Big(\big(1 + \operatorname{sign}(\sin(y))2\varepsilon_{rel}\big)\sin(y)\Big)\\
r &\geq \mathcal{R}_{-\infty}\Big(\big(1 - \operatorname{sign}(\sin(y))2\varepsilon_{rel}\big)\sin(y)\Big)
\end{aligned}
\tag{6}
$$

To mechanically eliminate the intermediate variable y — that is to derive bounds for r that are functions of x — is not trivial in cases where the functions are not monotonic. In such cases the difficulty lies in determining which of the bounds should be used. In contrast, (7) shows the equivalent information expressed using enclosures.

$$
\begin{aligned}
y &\in \mathcal{R}_{out}\Big((1 + [-2\varepsilon_{rel}, 2\varepsilon_{rel}])\cos(x)\Big)\\
r &\in \mathcal{R}_{out}\Big((1 + [-2\varepsilon_{rel}, 2\varepsilon_{rel}])\sin(y)\Big)
\end{aligned}
\tag{7}
$$

Since we know that y lies in a given enclosure and the enclosure for r is given in terms of y, we may replace y in the constraint to obtain a constraint on r involving only x. It follows that the two constraints in (7) compose to

$$
r \in \mathcal{R}_{out}\Big((1 + [-2\varepsilon_{rel}, 2\varepsilon_{rel}])\sin\Big(\mathcal{R}_{out}\Big((1 + [-2\varepsilon_{rel}, 2\varepsilon_{rel}])\cos(x)\Big)\Big)\Big)
$$

Notice that the argument to sin in this formula is an interval. We follow the set theoretic tradition that whenever $f : X \to Y$ is a function and $A \subseteq X$, then $f(A)$ is defined as

$$
f(A) \coloneqq \{f(a) \mid a \in A\}
$$

Since sin is a continuous function, the image of an interval — i. e. a connected set — is an interval.

This *compositionality* of enclosure inclusion constraints is one of two main advantages of using these constraints, the second being *conciseness* of the resulting annotations. What may seem to be an aesthetic argument, is in fact one found in engineering, which is that a clearly formulated specification is less likely to be misinterpreted.

2.2 Formalising enclosure constraints

The SPARK annotation language is essentially first order logic with variables over integers and reals. We wish to add primitives for handling enclosures and for relating

floating point expressions and enclosures. In the examples above we have used the constant intervals $[-\varepsilon_{abs}, \varepsilon_{abs}]$ and $[-\varepsilon_{rel}, \varepsilon_{rel}]$, which we shall denote by $\boldsymbol{\varepsilon}_{abs}$ and $\boldsymbol{\varepsilon}_{rel}$, respectively. They come in handy when expressing accuracy guarantees for the example program shown in Fig. 2 on the next page. We wish to provide a way of constructing enclosures from a pair of real functions, and for this purpose we define the function *hull* as follows:

Definition 1 (*hull*). *Let $x = [\underline{x}, \overline{x}]$ and $y = [\underline{y}, \overline{y}]$ be intervals. Then we define* hull *of x and y as the smallest interval including both x and y, i. e.*

$$hull(x, y) := \{z \in R \mid l \leq z \leq r\}$$

where

$$l = \min\{\underline{x}, \underline{y}\} \quad r = \max\{\overline{x}, \overline{y}\}$$

This definition is easily lifted to interval-valued functions. In other words, for any arbitrary set X and interval-valued function f and g over X, we define the interval-valued function $hull(f, g)$ pointwise using the formula

$$hull(f, g)(x) = hull(f(x), g(x))$$

In order to relate floating point expressions to enclosures, thus providing a way to express bounds on the values of program floating point variables, we have the predicate *ni*, taking a floating point expression e and an enclosure $E = [\underline{E}, \overline{E}]$ as arguments and defined by

$$ni(e, E) \equiv (\underline{E} \leq e) \wedge (e \leq \overline{E})$$

where we demand that the inequalities hold pointwise.

The SPARK implementation of our extension to the annotation language uses SPARK *proof functions*, which provide a way of defining function symbols that then may be used as annotation primitives. Proof functions are uninterpreted by the VC generator, which gives us control over the interpretation of our additional primitives. We represent the constant intervals $\boldsymbol{\varepsilon}_{abs}$ and $\boldsymbol{\varepsilon}_{rel}$ by the nullary functions `epsi_abs` and `epsi_rel`, respectively. *hull* is represented by the function `hull` and *ni* is represented by the predicate[3] `ni`. Since within annotations the operators `*`, `+` and `/` etc. correspond to the exact real operations, we use proof functions `add_fp`, `multiply_fp` and `divide_fp`, etc. to denote the rounded floating point operations of the default SPARK floating point type.

3 Example

We show how to use our extended language to annotate an example program in SPARK. As example we give an implementation of the square root function using Newton's algorithm and show that when executed on an IEEE compliant system, the program satisfies precision requirements similar to those given in the Ada Reference Manual [6]. Fig. 2 on the following page shows a SPARK core Ada implementation of the algorithm.

[3] Predicates may be viewed as boolean valued functions.

```
1 r := x;
2 s := 0.0;
3 while r /= s loop
4     s := r;
5     r := 0.5*(s+(x/s));
6 end loop;
```

Fig. 2. `My_Sqrt` implementation.

3.1 Specification

The weakest precondition for which the program behaves as expected is $x \geq 0$. However, to make the subsequent arguments easier we choose the stronger precondition

$$x \geq 2 \tag{8}$$

Moreover, by choosing a range of input values for which the computation is well behaved, we were able to easily find annotations that are very simple and yet prove the desired properties.

A reasonable contract for functional properties of the program should express that the returned floating point value is reasonably close to the exact square root. As mentioned in the preceding section, the Ada Reference Manual provides accuracy guarantees for implementations of elementary functions. Thus, our postcondition may be formulated as

$$\exists d \in [-2\varepsilon_{rel}, 2\varepsilon_{rel}] \,.\, r = \mathcal{R}((1+d)\sqrt{x}) \tag{9}$$

We have seen that it is useful to approximate such predicates using enclosures. In this case (9) becomes

$$r \in \mathcal{R}_{out}((1+2\boldsymbol{\varepsilon}_{rel})\sqrt{x}) \tag{10}$$

The precondition (8) assures us that no floating point value under consideration is denormalised or negative, which means that rounding may be represented by scaling

$$x \geq \varepsilon_{abs} \rightarrow \exists \varepsilon \in [-\varepsilon_{rel}, \varepsilon_{rel}] \,.\, \mathcal{R}(x) = (1+\varepsilon)x$$

which allows us to write (10) without the rounding operator as

$$r \in (1+\boldsymbol{\varepsilon}_{rel})(1+2\boldsymbol{\varepsilon}_{rel})\sqrt{x} \tag{11}$$

Due to the choice of precondition, we can actually use the following slightly tighter version of the constraint (11)

$$r \in (1+3\boldsymbol{\varepsilon}_{rel})\sqrt{x}$$

which allows us to write the following contract for the program in Fig. (2)

$$\texttt{pre}\ x \geq 2\ \texttt{post}\ r \in (1+3\boldsymbol{\varepsilon}_{rel})\sqrt{x} \tag{12}$$

We express the postcondition from (12) in the annotation language using the new primitives `ni` and `epsi_rel` and the exact real square root `sqrt`

$$\texttt{ni}(r, (1.0 + 3.0 * \texttt{epsi\_rel}) * \texttt{sqrt}(x))$$

```
1 --# pre x >= 2;
2 --# post ni(r,(1.0+3.0*epsi_rel)*sqrt(x));
3 r := x;
4 s := 0.0;
5 while r /= s loop
6     s := r;
7     --# assert ni(s,hull(1,x));
8     r := 0.5*(s+(x/s));
9 end loop;
```

Fig. 3. Annotated `My_Sqrt` implementation.

3.2 Loop invariant

When executing the program in exact real arithmetic, we observe that the values of s all fall within the interval $[1, x]$. It turns out that the condition

$$s \in [1, x] \tag{13}$$

suffices for showing that no division by zero occurs, even when taking rounding errors into consideration. We formulate (13) in the annotation language using the new primitives `ni` and `hull`

$$\texttt{ni}(s, \texttt{hull}(1.0, x))$$

In Fig. 3 above we show the program code from Fig. 2 annotated with the contract (12) and the loop invariant (13).

3.3 Validation

In this section we prove partial correctness of the example program with respect to the contract (12), as described in the section above. The VC generator extracts the correctness theorem from the annotated program by deriving verification conditions for paths in the program's control flow graph. The graph is cut at points defined by the annotations, thus creating a set of paths for which VCs are obtained by weakest precondition calculus. Once generated, the VCs are simplified using a symbolic automated theorem prover by applying a number of simple symbolic transformation and inference rules.

The verification conditions generated from the annotated program are shown in Figs. 4 to 8 on the following page. Note the appearance of the functions `multiply_fp`, `add_fp` and `divide_fp` in Figs. 5 and 8. They are needed because currently SPARK treats floating point arithmetic as exact real arithmetic. Therefore, function wrappers corresponding to floating point operators are used to ensure that rounding is accounted for when deriving the VCs. Thus, `add_fp`, `multiply_fp` and `divide_fp` are wrappers for the floating operations $\oplus$, $\otimes$ and $\oslash$, respectively. The functions are not interpreted by the VC generator, which leaves us to handle the functions in external tools as we see fit.

```
H1: x >= 2 .
    ->
C1: ni(x, hull(1, x)) .
```

Fig. 4. Verification condition for path(s) from start to assertion of line 7.

```
H1: ni(s, hull(1, x)) .
H2: x >= 2 .
H3: s <> 0 .
H4: multiply_fp(1 / 2, add_fp(s, divide_fp(x, s))) <> s .
    ->
C1: ni(multiply_fp(1 / 2, add_fp(s, divide_fp(x, s))),
      hull(1, x)) .
```

Fig. 5. Verification condition for path(s) from assertion of line 7 to assertion of line 7.

```
H1: ni(s, hull(1, x)) .
H2: x >= 2 .
    ->
C1: s <> 0 .
```

Fig. 6. Verification condition for path(s) from assertion of line 7 to precondition check associated with statement of line 8.

```
*** true .  /* contradiction within hypotheses. */
```

Fig. 7. Verification condition for path from start to finish assuming the loop not entered.

```
H1: ni(s, hull(1, x)) .
H2: x >= 2 .
H3: s <> 0 .
H4: multiply_fp(1 / 2, add_fp(s, divide_fp(x, s))) = s .
    ->
C1: ni(multiply_fp(1 / 2, add_fp(s, divide_fp(x, s))),
      (1 + 3 * epsi_rel) * sqrt(x)) .
```

Fig. 8. Verification condition for path(s) from assertion of line 7 to finish.

Proofs of verification conditions are performed "by hand" and we address the task of automating the proofs in Sect. 4 on the next page.

The proof of the VC for the path from start to the annotation on line 7, shown in Fig 4, follows directly from the definition of *hull*, found in Sect. 2.2.

The proof of the VC for the path around the loop, shown in Fig 5, is straightforward. It suffices to show the implication

$$(s \in [1, x] \wedge x \geq 2) \rightarrow \frac{1}{2} \otimes (s \oplus (x \oslash s)) \in [1, x]$$

We do this by showing that the maximum and minimum of $\frac{1}{2} \otimes (s \oplus (x \oslash s))$, for $s \in [1, x]$ and $x \geq 2$, both lie inside $[1, x]$. The hypotheses imply that all floating point arguments in the expression are normalised, whence there are $\varepsilon_1, \varepsilon_2, \varepsilon_3 \in \boldsymbol{\varepsilon}_{rel}$, such that the conclusion becomes

$$(1 + \varepsilon_1)\frac{1}{2}(1 + \varepsilon_2)(s + (1 + \varepsilon_3)\frac{x}{s}) \in [1, x] \tag{14}$$

which follows from $s \in [1, x] \wedge x \geq 2$ by direct computation.

The proof of the VC for the path from the annotation on line 7 to the precondition check on line 8, shown in Fig 6, trivially follows from the hypothesis $s \in [1, x]$. This proves that no division by zero exceptions can result from running our program, when called with values ≥ 2.

The proof of the VC for the path from start to finish, without entering the loop, shown in Fig. 7, was performed by the VC simplifier. It detected that for this path $x = 0$ must hold, which conflicts with the precondition hypothesis $x \geq 2$.

The proof of the VC for the path from the annotation on line 7 to finish, shown in Fig. 8, is straightforward. By the argument made when deducing (14) we find the following sufficient condition

$$(1 + \varepsilon_1)\frac{1}{2}(1 + \varepsilon_2)(s + (1 + \varepsilon_3)\frac{x}{s}) = s \rightarrow s \in (1 + 3\boldsymbol{\varepsilon}_{rel})\sqrt{x} \tag{15}$$

which follows from $s \in [1, x] \wedge x \geq 2$ by direct computation.

3.4 Generalisation

So far in this section we have focused on one specific implementation of the square root function, i. e. that based on the Newton-Raphson method. This is essentially finding the (non-negative) zero of the function f_x defined by $f_x(z) = z^2 - x$ to obtain the square root of the (non-negative) real number x.

The Newton-Raphson method of zero finding is known to converge quadratically under fairly general conditions [8]. In the case of the program in Fig. 2, if at a certain

stage n correct digits of the result — i. e. the square-root of x — have been calculated, then after the next iteration of the loop at least $2n - 1$ correct digits will be obtained.

One would expect to get similar properties for any function defined using Newton-Raphson method as long as the conditions for quadratic convergence are met. This in no way is true. As an example consider the *Lambert W* function [9]:

Definition 2 (Lambert *W* Function). *Consider the function* $f(z) = z\,e^z$ *defined over the set of complex numbers. The inverse of this function is called* Lambert *W function, i. e.* $f(W(z)) = z$*. In general this is a multivalued complex function. Over the real numbers, by Lambert W function we mean the single-valued function defined over the set* $\{r \in \mathbb{R} \mid r \geq e^{-1}\}$.

To modify the code of Fig. 2 to suit the computation of *W* function, one would hope that changing line 5 to

$$\texttt{r} := (\texttt{s} * \texttt{s} + \texttt{x}/\texttt{Exp(s)})/(1.0 + \texttt{s});$$

would suffice as this is the formula obtained by applying Newton-Raphson method.

However, the termination condition of line 3 does not work anymore. Even though in both cases the conditions are satisfied for quadratic convergence, there are still at least two practical issues. First of all, quadratic rate of convergence is just the description of a convergence class, whilst in practice implementations have their own characteristic behaviour. Secondly, in floating point computations, round-off and truncation errors come into play. Thus, in some cases the computed values of `r` and `s` may oscillate between two — or in general jump over distinct pairs of elements from a finite set of — floating point numbers. For instance, if the code uses the Ada data type `Float` then with number 8 assigned to the input x the oscillation can be observed.

Therefore, the condition of line 3 may need to be replaced by a more theoretically sound condition such as

$$\texttt{while } |r - s| > p \texttt{ loop}$$

using a specific p for *precision*. However with this formulation, VCs will not be as neat as those obtained for square root implementation.

4 Automation of Proofs

To automate the proof of VCs such as those we obtained in our example, we could use symbolic techniques, trying to emulate the manual proofs shown earlier. We do not pursue this approach here. Instead our approach is based on evaluating the inequalities and enclosure constraints contained in our VCs with the goal of proving that the conditions are true by some safe margin.

4.1 Interval evaluation for proving inequalities

Such evaluation is often performed using interval arithmetic, assigning to each variable its interval range and hoping that after interval evaluation the safe estimates of the expressions' values are separated. This helps prove an inequality over the whole range at

once. More formally, what this process does can be described as follows: assume that we want to decide whether

$$\forall \boldsymbol{x} \in D : f(\boldsymbol{x}) > g(\boldsymbol{x}) \tag{16}$$

where $D = D_1 \times \ldots \times D_n$ is an n-dimensional rectangle in $\mathbb{R}^n$. Moreover, let f and g be expressions that use only:

- the variables $\boldsymbol{x} = x_1, \ldots, x_n$
- constant intervals with rational endpoints (including singletons)
- exact versions of the operators $+, -, \times, /, \sqrt{\cdot}$
- rounded versions of the above operators: $\oplus, \ominus, \otimes, \oslash, \circ\sqrt{\cdot}$

A naive attempt to prove (16) is to replace each variable x_i with its range D_i and evaluate the expression using correctly rounded interval arithmetic. The rounded operators additionally widen the resulting interval according to some ε-bounds for the rounding error of the operators. This widening is similar to the effect of the $\mathcal{R}_{out}$ operator used in Subsection 2.1. Such an evaluation may suffer from inaccuracies arising from *dependency* problems. For instance, it may lead to very large intervals whenever one variable is used several times, as in the expression $s \oplus (x \oslash s)$.

In our earlier paper [4] we showed how dependency problems are successfully tackled using polynomial function enclosure arithmetic instead of interval arithmetic. In this approach each variable x_i is replaced by the projection thin enclosure $\lambda(y_1, \ldots, y_n).[y_i, y_i]$ and the operations are evaluated using uniformly outwards-rounded pointwise operations on function enclosures.

We can illustrate the inequality decision process and its possible outcomes as shown in Fig. 9 on the following page. In the figure we refer to the structure $\mathcal{I}(D \to \mathbb{R})$. This is a continuous domain [10] comprising all function enclosures formed by pairs of continuous functions $[\underline{h}, \overline{h}]$ with $\underline{h} \leq \overline{h}$ and ordered by *enclosure inclusion*:

$$[\underline{h}_1, \overline{h}_1] \sqsubseteq [\underline{h}_2, \overline{h}_2] \iff \underline{h}_2 \leq \underline{h}_1 \text{ and } \overline{h}_1 \leq \overline{h}_2$$

We call $\mathcal{I}(D \to \mathbb{R})$ the *interval function domain* over the region D.

We can illustrate the inequality decision process and its possible outcomes as shown in Fig. 9.

4.2 Interval enclosure evaluation for proving enclosure constraints

One way to show that an enclosure constraint $f \in g$ is true — where f is a floating point expression and g is an enclosure such as those in our VCs — is to find an *outer approximation* $f^{\downarrow}$ of f and show that $f^{\downarrow} \sqsubseteq g$. To do this in practice, we need to find an *inner approximation* $g^{\uparrow}$ of g and then show $f^{\downarrow} \sqsubseteq g^{\uparrow}$. This process is illustrated in Fig. 10. An inner approximation of an enclosure is not always consistent, i. e. the "upper" bound is not always above the "lower" bound. In fact, for a thin enclosure, such as f in Figs. 10 and 11, the inner approximation is almost always inconsistent at all points of the domain D. To account for such inconsistent enclosures correctly, we need to switch from the interval domain $\mathcal{I}(D \to \mathbb{R})$ to an *interval lattice*, which we denote $\mathcal{IL}(D \to \mathbb{R})$. The lattice $\mathcal{IL}(D \to \mathbb{R})$ is very similar to $\mathcal{I}(D \to \mathbb{R})$ except that

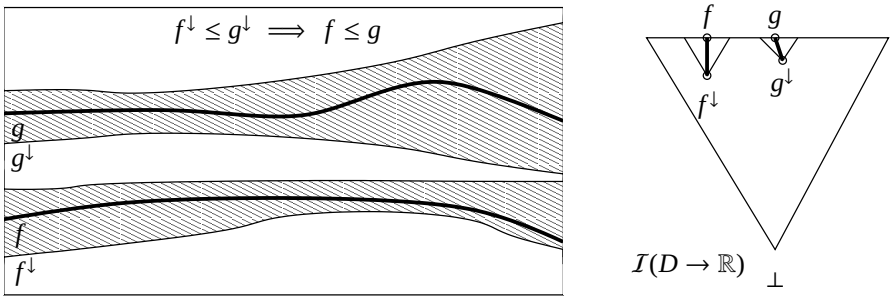

Fig. 9. Deciding an inequality by approximate evaluation

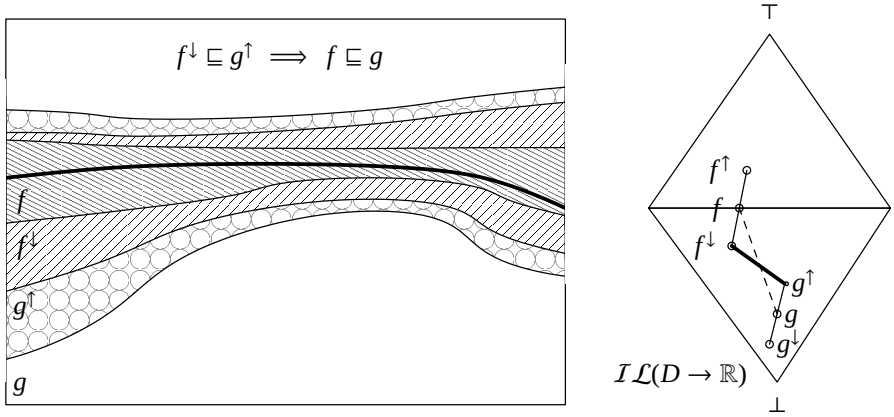

Fig. 10. Deciding an enclosure constraint by approximate evaluation: proved true

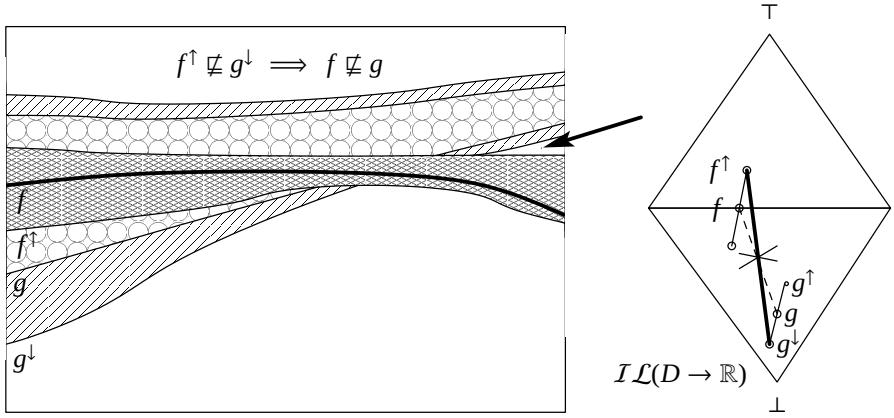

Fig. 11. Deciding an enclosure constraint by approximate evaluation: proved false

its elements $[\underline{h}, \overline{h}]$ do *not* have to satisfy $\underline{h} \leq \overline{h}$, i. e. they can be *inconsistent*. The order in the lattice is defined in exactly the same way as in the domain [11].

The inconsistent elements can play an explicit positive role when attempting to *disprove* the enclosure constraint $f \in g$. If one succeeds in showing that $f^{\uparrow} \not\sqsubseteq g^{\downarrow}$, then it follows that $f \notin g$, as illustrated in Fig. 11. Recall that the "enclosure" $f^{\uparrow}$ is inconsistent, i. e. the boundary above f is formally the lower bound of $f^{\uparrow}$. Thus the area highlighted by the big arrow is an evidence that the *lower* bound of $g^{\downarrow}$ is not fully below the *lower* bound of $f^{\uparrow}$, which means that $f \not\sqsubseteq g$, which another way of saying $f \notin g$.

4.3 Towards a complete prover

An obvious problem with the evaluation approach to proving as we just discussed is that it is incomplete, i. e. many inequalities and enclosure constraints will fail to get decided. We therefore make the evaluation a part of a more complex process to gain as much completeness as possible.

Our approach is based on a *branch-and-prune* algorithm [12] using polynomial enclosures [4] to approximate the real expressions and enclosures appearing in the VCs. The algorithm attempts to decide the terms in the VC by approximating each real expression with polynomial upper and lower bounds. Then a decision procedure for polynomial inequalities is used to decide if the term is true or false for all possible values of the variables. If the decision procedure fails to decide a term — and as a consequence also the VC — then the domain of a variable is bisected and the approximation step is repeated. Since a smaller domain leads to tighter approximations, there is hope that the VC will eventually either be proved or shown to be false over a sub box of the initial domain box.

As mentioned above, this type of algorithm uses *outer approximations* when constructing enclosures for expressions. As a consequence of making these approximations we lose the ability to prove inclusions such as $X \subset X$. A special case occurs when the boundaries of the two enclosures are touching, say as in $\forall x \in [0, \frac{\pi}{4}]. \sin(x) < \cos(x)$. No silver bullet exists for this pathology and in practice one will almost certainly encounter VCs having terms with touching boundaries. We do however hope to eliminate some cases by symbolic pre-processing of VCs before the numeric solving.

One way of addressing the loss of precision resulting from the approximation steps mentioned above is to employ an exact decision procedure for polynomial inequalities. One such approach is taken by Paulson and Akbarpour [13] who prove strict inequalities between real expressions by approximating the expressions with polynomials and then passing the polynomial inequalities to a decision procedure using partial cylindrical algebraic decompositions [14]. The decision procedure used is doubly exponential in the number of variables, so enclosure constraints could — due to their compositional nature — produce VCs with a comparatively small number of variables, possibly making automation of proofs using Paulson's approach feasible.

Experience has shown that, with the current state of automated real number theorem provers, full proof automation for VCs arising from floating point code is not practically feasible. Therefore, a hybrid approach combining automated and interactive proving techniques is necessary. There is hope for considerable synergy as each ap-

proach complements the other. Interactive proving is suitable for equational reasoning, while automated numeric techniques are suitable for reasoning about inequalities.

5 Conclusions

We have proposed an extension to the SPARK annotation language which enables us to express formulae denoting semantic properties of floating point computation in terms of enclosure constraints.

We have implemented an example program in SPARK and annotated it with the extended language demonstrating how enclosure constraints shorten the annotations. This in turn makes them easier to read and consequently reduces the risk of erroneous use of the program. VCs for partial correctness of the program with respect to numerical accuracy guarantees and division by zero exception freedom were derived and simplified automatically by the current SPARK tools without need for modification.

We have also demonstrated that enclosure constraints compose and thus eliminate redundant intermediate variables of a computation. This is a very important property, as automated proving always suffers from severe limitations on the number of variables appearing in a theorem.

Our approach can fit comfortably in a domain/lattice theoretic framework as alluded to in subsection 4.2. Almost all the procedures and conclusions have direct and neat interpretation in those semantic frameworks.

6 Further Work

Future research will be focusing on developing a number of realistic programs, annotated with the proposed language, in order to obtain a test suite of VCs for functional properties and absence of numerical exceptions which will serve as benchmarks for a dedicated solver.

The solver will be based on an earlier numerical constraint solver [4]. The main addition will be an implementation of inward approximating enclosure arithmetic and added support for the extension of the SPARK annotation language proposed above.

As pointed out in Sect. 4, we will address the problem of touching boundaries implementing a symbolic pre-processing stage preceding the numerical solving.

On a more theoretical level, one possible issue to address would be the issue of *complexity*. We have observed an increase in efficiency by moving from piece-wise constant enclosures to polynomial enclosures. However, we believe that there is an optimum degree for the class of polynomials used above which, the burden of handling the polynomials (symbolically) exceeds the gain in convergence speed.

There is also further room to investigate the link with domain theory and the lattices arising from Dedekind reals as in [11].

References

1. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: Design and implementation of a special-purpose static program analyzer for safety-

critical real-time embedded software. In: The essence of computation: complexity, analysis, transformation. Springer-Verlag New York, Inc., New York, NY, USA (2002) 85–108
2. Granvilliers, L., Benhamou, F.: Algorithm 852: RealPaver: An interval solver using constraint satisfaction techniques. ACM Transactions on Mathematical Software **32**(1) (2006)
3. Ratschan, S., et al.: RSolver. `http://rsolver.sourceforge.net` (2004) Software Package.
4. Duracz, J.A., Konečný, M.: Polynomial function enclosures and floating point software verification. In: Constraints in Formal Verification/IJCAR 2008. (2008) 56–67
5. Barnes, J.: High Integrity Software: The SPARK Approach to Safety and Security. 2 edn. Addison-Wesley (April 2003)
6. ISO: Ada Reference Manual, ISO/IEC 8652:2007(E) Ed. 3. (2007)
7. Society, I.C.: IEEE Std 754-1985. Institute of Electrical and Electronics Engineers, New York (1985)
8. Kunz, K.S.: Numerical Analysis. McGraw-Hill Book Company (1957)
9. Hayes, B.: Why W? American Scientist **93**(2) (March–April 2005) 104–108
10. Gierz, G., Hofmann, K.H., Keimel, K., Lawson, J.D., Mislove, M.W., Scott, D.S.: Continuous Lattices and Domains. Volume 93 of Encycloedia of Mathematics and its Applications. Cambridge University Press (2003)
11. Taylor, P.: A lambda calculus for real analysis. In Grubba, T., Hertling, P., Tsuiki, H., Weihrauch, K., eds.: Computability and Complexity in Analysis. Number 326 in Informatik Berichte, FernUniversität in Hagen (2005) 227–266 revised version at www.PaulTaylor.EU/ASD/lamcra/.
12. Vu, X.H., Silaghi, M.C., Sam-Haroud, D., Faltings, B.: Branch-and-prune search strategies for numerical constraint solving. http://arxiv.org/abs/cs/0512045 (2005) Accessed on 3rd September 2007.
13. Akbarpour, B., Paulson, L.C.: Metitarski: An automatic prover for the elementary functions. In Autexier, S., Campbell, J., Rubio, J., Sorge, V., Suzuki, M., Wiedijk, F., eds.: AISC/MKM/Calculemus. Volume 5144 of Lecture Notes in Computer Science., Springer (2008) 217–231
14. Brown, C.W.: QEPCAD B: a system for computing with semi-algebraic sets via cylindrical algebraic decomposition. SIGSAM Bull. **38**(1) (2004) 23–24