Automatic Generation of Staged Geometric Predicates

Aleksandar Nanevski
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891
aleks@cs.cmu.edu

Guy Blelloch
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891
blelloch@cs.cmu.edu

Robert Harper
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891
rwh@cs.cmu.edu

ABSTRACT

Algorithms in Computational Geometry and Computer Aided Design are often developed for the Real RAM model of computation, which assumes exactness of all the input arguments and operations. In practice, however, the exactness imposes tremendous limitations on the algorithms – even the basic operations become uncomputable, or prohibitively slow. When the computations of interest are limited to determining the sign of polynomial expressions over floating-point numbers, faster approaches are available. One can evaluate the polynomial in floating-point first, together with some estimate of the rounding error, and fall back to exact arithmetic only if this error is too big to determine the sign reliably. A particularly efficient variation on this approach has been used by Shewchuk in his robust implementations of Orient and InSphere geometric predicates.

We extend Shewchuk's method to arbitrary polynomial expressions. The expressions are given as programs in a suitable source language featuring basic arithmetic operations of addition, subtraction, multiplication and squaring, which are to be perceived by the programmer as exact. The source language also allows for anonymous functions, and thus enables the common functional programming technique of *staging*. The method is presented formally through several judgments that govern the compilation of the source expression into target code, which is then easily transformed into SML or, in case of single-stage expressions, into C.

1. INTRODUCTION

Algorithms in Computational Geometry and Computer Aided Design are often created for the Real RAM model of computation. Real RAM model assumes exactness of all the arguments and operations involved in the calculations, thus making it easy to carry out the mathematical arguments behind the algorithm. Unfortunately, this very fact implies that the computations have to be done with unbounded or infinite precision, which can render the basic operations and predicates prohibitively slow or even uncomputable.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICFP'01, September 3-5, 2001, Florence, Italy.
Copyright 2001 ACM 1-58113-415-0/01/0009 ...$5.00.

A practical and very useful compromise, when applicable, is to assume that the input arguments are of floating-point type. It is also very common that the required functionality involves computation of only the *sign* of a given *polynomial* expression. Such calculations are, for example, used in the geometric predicates for determining whether a point is in/out/on a given line, circle, plane, sphere... These predicates are, in turn, fundamental building blocks of algorithms for some basic geometric structures such as convex hulls and Delaunay triangulations.

However, floating-point alone is not sufficient to guarantee that the evaluation of a polynomial expression will correctly obtain its sign. The rounding error accumulated during the computation, if sufficiently large, can perturb and change the final result. If this computation is part of a geometric algorithm, it can present the program with an inconsistent view of the data set, and cause it to produce incoherent results, diverge, or even crash. On the other hand, under the stated assumptions, the exact sign can always be computed, albeit slowly, by first converting the floating-point arguments into rational numbers, and then carrying out the prescribed operations in rational arithmetic.

One method that has been proposed as an efficiency improvement to the exact rational arithmetic involves the use of *floating-point filters*. A floating-point filter carries out the given computation in floating-point first, together with some sort of estimate of the rounding error, and falls back to exact arithmetic only if the estimated error is too big to reliably determine the sign [1, 2, 6, 3, 9]. Thus, it "filters out" the easy computations whose sign can be quickly determined and only leaves the hard ones for the exact arithmetic. A particularly efficient variation of this approach has been described by Jonathan Shewchuk in his PhD thesis [11]. Aside from performing the floating-point part of the computation as the first phase of the filter, it introduces additional filtration phases of ever-increasing precision. The phases are attempted in order, each phase building on the result from the previous one, until the correct sign is obtained.

There are two difficulties related to Shewchuk's method that this paper addresses:

1. Developing robust geometric predicates in this style can be very cumbersome and error prone. For example, the basic InSphere predicate which tests whether a point is in/out/on the sphere determined by three other points is represented by a simple 4×4 matrix. However, Shewchuk's implementation of InSphere consists of about 580 lines of C code. In addition, one needs to perform the error analysis of the given poly-

nomial expression, which is also a tedious procedure. A solution is to automate this process by using an expression compiler [2, 5]. However, to the best of our knowledge, none of the existing expression compilers is capable of performing the analysis required by the multi-phase floating-point filters.

2. We are also interested in designing predicates for functional languages and exploiting the common functional programming technique of staging to speed up the computation. For example, consider filtering a set of points to see on what side of a plane defined by three points they lie. The test can be staged by first forming the plane and then checking the position of each point from the set. This obviates the need to repeat the part of the computation pertinent to the plane whenever a new point is tested, and can potentially save a lot of work. Such staging of programs is naturally exploited in functional programming languages, but unfortunately, the expression compilers available to date work only with C.

This paper reports on an expression compiler that handles these shortcomings. The input to the compiler is a function written in an appropriate source language offering the basic arithmetic operations of addition, subtraction, multiplication and squaring, and allowing for nested anonymous functional expressions (*stages*). All the operations in the source language are perceived as exact. The output of the compiler is a program in the target language designed to be easily converted into Standard ML (SML) or, in the case of single-stage programs, to C. The resulting SML or C program will determine the *sign* of the source function at the given floating-point arguments, using a floating-point filter with several *phases*, when exact computation needs to be performed. In particular, in the case of Shewchuk's basic geometric predicates, the expression compiler will generate code that, to a considerable extent, reproduces that of Shewchuk.

The rest of the paper is organized as follows. Section 2 summarizes the main ideas behind floating-point filters and arbitrary precision floating-point arithmetic. The source and target languages are presented in Section 3, and the program transformation process is described in Section 4. Performance comparison with Shewchuk's predicates is given in Section 5 and a definition of selected judgments governing the compilation follows in the Appendix.

2. BACKGROUND

From here on we assume floating-point arithmetic as prescribed by the IEEE standard and the to-nearest rounding mode with the round-to-even tie-breaking rule [4]. We also assume that no overflows or underflows occur.

One of the most important properties of a floating-point arithmetic is the correct rounding of the basic arithmetic operations. It requires that the computed result always look as if it were first computed exactly, and then rounded to the number of bits determined by the precision of the arithmetic. If x and y are floating-point numbers, $\circledast$ is the "rounded" floating-point version of the operation $* \in \{+, -, \times\}$, and $x \circledast y$ is a floating-point number with a normalized mantissa (i.e. is not a denormalized number), a consequence of the correct rounding is that

$$|x * y - x \circledast y| \leq \epsilon |x \circledast y| \quad \text{and} \quad |x * y - x \circledast y| \leq \epsilon |x * y|$$

The quantity ϵ in the above inequality is called "machine epsilon". If m is the precision of the arithmetic, i.e. the number of bits reserved for the normalized mantissa (without the hidden leading bit), then $\epsilon = 2^{-(m+1)}$. In the IEEE standard for double precision, for example, $\epsilon = 2^{-53}$. By abuse of notation, the above inequalities are often stated respectively as

$$x * y = (1 \pm \epsilon)(x \circledast y) = x \circledast y \pm \epsilon |x \circledast y| \tag{1}$$

and

$$x * y = x \circledast y \pm \epsilon |x * y|$$

The equation (1) provides a bound on absolute error of the expression when the expression consists of only a single floating point operation. Notice that the error is composed of two multiples, ϵ and $|x * y|$, the first of which does not depend on the arguments x and y. The rounding error for a *composite* expression can also be split into two multiples, one of which does not depend on the arguments of the expression. This part of the error need not be computed in runtime when all the arguments of the expression are supplied, but can rather be completely obtained while preprocessing the expression. To this end, assume that the exact values X_i are approximated in floating-point as x_i with absolute error $\delta_i p_i$, i.e. that for $i = 1, 2$ we have

$$X_i = x_i \pm \delta_i p_i$$

Assume in addition that the quantities δ_i do not depend on any run-time arguments and that the invariant $|x_i| \leq p_i$ holds. This is clearly true in the base case when x_i is obtained from a single operation on exact arguments, as can be seen from (1) where $\delta_i = \epsilon$ and $p_i = |x_i|$. The quantities δ_i are rational numbers, and the values p_i are floating-point. Diverging slightly from the customary nomenclature, we call these two multiples respectively the "relative error" and the "permanent" of the approximation x_i.

Using the inequalities for rounded floating-point arithmetic from above, we can derive

$$\begin{aligned}
&|(X_1 + X_2) - (x_1 \oplus x_2)| \;= \\
&\quad = \; |(X_1 + X_2) - (x_1 + x_2) + (x_1 + x_2) - (x_1 \oplus x_2)| \\
&\quad \leq \; |(X_1 + X_2) - (x_1 + x_2)| + |(x_1 + x_2) - (x_1 \oplus x_2)| \\
&\quad \leq \; (\delta_1 p_1 + \delta_2 p_2) + \epsilon |x_1 \oplus x_2| \\
&\quad \leq \; \max(\delta_1, \delta_2)(p_1 + p_2) + \epsilon (p_1 \oplus p_2) \\
&\quad \leq \; \max(\delta_1, \delta_2)(1 + \epsilon)(p_1 \oplus p_2) + \epsilon (p_1 \oplus p_2) \\
&\quad = \; \Big(\epsilon + \max(\delta_1, \delta_2)(1 + \epsilon)\Big)(p_1 \oplus p_2)
\end{aligned}$$

The above inequality is, by abuse of notation, customarily written as

$$X_1 + X_2 = x_1 \oplus x_2 \pm \Big(\epsilon + \max(\delta_1, \delta_2)(1 + \epsilon)\Big)(p_1 \oplus p_2)$$

The relative error of the composite expression $X_1 + X_2$ is then $\epsilon + \max(\delta_1, \delta_2)(1 + \epsilon)$ and its permanent is $p_1 \oplus p_2$. Notice that the relative error again does not depend on the run-time arguments, and that the invariant $|x_1 \oplus x_2| \leq p_1 \oplus$

p_2 is preserved. Similar derivations produce

$$\begin{aligned} X_1 - X_2 &= x_1 \ominus x_2 \pm \Big(\epsilon + \max(\delta_1, \delta_2)(1+\epsilon)\Big)(p_1 \oplus p_2) \\ X_1 X_2 &= x_1 \otimes x_2 \\ &\quad \pm \Big(\epsilon + (\delta_1 + \delta_2 + \delta_1\delta_2)(1+\epsilon)\Big)(p_1 \otimes p_2) \qquad (2) \\ X_1^2 &= x_1 \otimes x_1 \pm \Big(\epsilon + (2\delta_1 + \delta_1^2)(1+\epsilon)\Big)(p_1 \otimes p_1) \end{aligned}$$

The above formulas provide a quick test for the sign of X_i. Obviously, x_i and X_i have the same sign if $|x_i| > \delta_i p_i$. However, this test is not completely satisfactory since it contains exact multiplication of a rational number δ_i and a floating-point number p_i. A simpler, although less tight test is

$$|x_i| > \lceil (1+\epsilon)\delta_i \rceil_{fp} \otimes p_i \qquad (3)$$

where $\lceil Q \rceil_{fp}$ denotes the smallest floating-point value above the rational number Q. This is indeed the inequality we use in our expression compiler to test the sign of an evaluated expression.

Another important feature of round-to-nearest arithmetic complying with the IEEE standard is that the roundoff error of the basic operations is always representable as a floating-point number and can be recovered from the result and the arguments of the operation.

THEOREM 1 (KNUTH). *In floating-point arithmetic with precision $m \geq 3$, if $x = a \oplus b$ and $c = x \ominus a$ then $a + b = x + ((a \ominus (x \ominus c)) \oplus (b \ominus c))$.*

Knuth's theorem is significant since it provides a way to quickly perform exact addition of two floating-point numbers. First the addition $x = a \oplus b$ is performed approximately, and then the roundoff error $e = (a \ominus (x \ominus c)) \oplus (b \ominus c)$ is recovered. This takes only 6 floating-point operations, which is generally much faster than first converting a and b into rational numbers, and then adding them in rational arithmetic. The two values (x, e) put together represent the exact sum of a and b. One can view this pair as a sparse representation of the sum in a digit system with a radix 2^{m+1}. Closing up the set of sparse representations under addition leads to a very efficient data structure for exact computation. The values of this data type are lists of floating-point numbers sorted by magnitude and satisfying certain technical conditions about the alignment of their mantissas. These lists are called *expansions*, and each expansion represents the *exact* sum of its elements. For the sake of illustration, here we only picture the process of adding a floating-point number to an expansion (Figure 1) and of summing up two expansions (Figure 2). Quick algorithms for other basic arithmetic operations on this data type have been devised as well [8, 11].

Another consequence of Knuth's theorem is a convenient ordering of operations that makes it possible to separate the computation into a sequence of filtering phases. Each phase is attempted after the previous one had failed to determine the sign reliably, and each computes with increasing precision, building on the result of the previous one.

The following example, although admittedly a bit contrived, is illustrative. Consider the expression $X = (a_x - b_x)^2 + (a_y - b_y)^2$ where a_x, a_y, b_x and b_y are floating-point values. To find the sign of X, let $v_x = a_x \ominus b_x$ and $v_y = a_y \ominus b_y$, and let e_x and e_y be the roundoffs from the

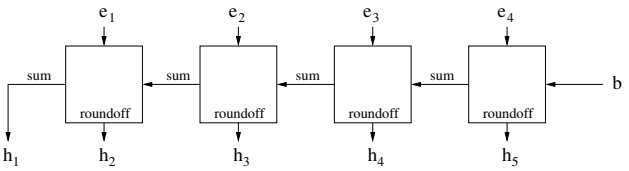

Figure 1: Adding a floating-point number to an expansion. The float b is added to the expansion $e_1+e_2+e_3+e_4$, to produce a new expansion $h_1+\cdots+h_5$.

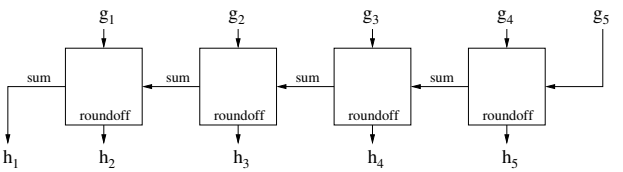

Figure 2: Summing two expansions. The components of $e_1 + e_2 + e_3$ and $f_1 + f_2$ are first merged by decreasing magnitude into the list $[g_1, \ldots, g_5]$, which is then "normalized" into an expansion $h_1 + \cdots + h_5$.

two subtractions. Then

$$\begin{aligned} X &= (v_x + e_x)^2 + (v_y + e_y)^2 \\ &= (v_x^2 + v_y^2) + (2v_x e_y + 2v_y e_x) + (e_x^2 + e_y^2) \end{aligned}$$

In this sum, the summand $v_x^2 + v_y^2$ is dominant, since $|e_x| \leq \epsilon|v_x|$ and $|e_y| \leq \epsilon|v_y|$ by (1). It is then a good heuristic to first compute $v_x^2 + v_y^2$ and test it for sign before proceeding, because it is likely that $v_x^2 + v_y^2$ will already have the same sign as X. The process can be sped up even more if this expression is first computed approximately to obtain $X^A = (v_x \otimes v_x) \oplus (v_y \otimes v_y)$. Then only if X^A has too big an error bound, as determined by the test (3), the computation of $X^B = v_x^2 + v_y^2$ is undertaken exactly using the data type of expansions. If X^B is also too crude an approximation of X, we can correct it by adding up the smaller terms $(2v_x e_y + 2v_y e_x)$, first approximately, and then correctly. Finally, if all of these approximations fail to give an answer, we can compute the exact result by adding the last summand $e_x^2 + e_y^2$ to the expansion computed in the previous phase. Using this approach, we will compute the exact value only if absolutely necessary, and even then, the efforts spent on previous phases will not be wasted, but will rather be reused to obtain the exact result in an efficient way.

This idea to generalize floating-point filters into a hierarchy of adaptive precision filtering phases is due to Shewchuk. While the number and type of adaptive phases, strictly speaking, can vary with the expression, his experiments pointed to a scheme with four phases as the optimal in practice for the basic geometric predicates that he considered. We adopt this scheme and present its formalization in Section 4. The arbitrary precision floating-point arithmetic and the data type of expansions is invented by Priest, and further optimized by Shewchuk. Detailed description and analysis of adaptive precision arithmetic and of the algorithms for basic operations can be found in their respective PhD theses ([8] and [11]).

The whole method described above relies on the fact that

```
phrases          φ ::= x | c | e
expressions      e ::= φ1 + φ2 | φ1 - φ2 | φ1 × φ2
                     | ~φ | sq φ
assignment lists α ::= val x = e | val x = e α
programs         π ::= fn [x1, ... , xn] => let α end
                     | fn [x1, ... , xn] => let α π end
```

Figure 3: Source language

$$\text{orient2}(A,B,C) = \text{sgn}\begin{vmatrix} a_x - c_x & b_x - c_x \\ a_y - c_y & b_y - c_y \end{vmatrix}$$

```
fn [ax, ay, cx, cy] =>
let val acx = ax - cx
    val acy = ay - cy

    fn [ bx, by ] =>
    let val d = acx × ( by - cy ) -
                acy × ( bx - cx )
    end
end
```

Figure 4: Orient2D predicate: definition and implementation in the source language.

the required floating-point operations will execute without any exceptions, i.e. that neither overflow nor underflow will occur during the computation. If exceptions do happen, the expansions holding the exact intermediate values may lose bits of precision and produce a distorted answer. A possible solution in such cases is to rerun the computation in some other, slower, form of exact arithmetic (for example in infinite precision rational numbers).

3. SOURCE AND TARGET LANGUAGES

The source language of the compiler is shown in Figure 3. Its syntax supports the basic arithmetic operations (including squaring), assignments and staged functional expressions. The arguments of the functions should be perceived as floating-point values, while the intermediate results are assumed to be computed exactly. Squaring is included among the arithmetic operations because it can often be executed quicker than the multiplication of two equal exact values, and has a better error bound. In addition, it provides the compiler with the knowledge that its result is non-negative, which can be used in some cases to optimize the code. In order to simplify the compilation process, the source language requires that all the assignments are non-trivial, i.e. it disallows assignments of variables or constants. A function defined in the source language is designed to compute the *sign* of the last expression in the assignment list of the function's last stage. Of course, a staged source function can be partially instantiated with an appropriate subset of the arguments in order to return a new function that encodes the rest of the computation. The source language does not have any syntactic constructs for the sgn function, but this function is always assumed at the last assignment of the last stage of the program.

As an example, consider the Orient2D geometric predicate and its implementation in Figure 4. Orient2D determines the position (in/out/on) of point $B = (b_x, b_y)$ with respect to the line from $A = (a_x, a_y)$ to $C = (c_x, c_y)$. The implementation in Figure 2 is staged in the coordinates of A and C. Once the predicate is applied to these two points, its result is a new function specialized to compute relative to the line $\overline{AC}$, without recomputing the intermediate results `acx` and `acy`.

```
reals            r ::= x | c | r1 ∗ r2 | r1 ⊛ r2 | sq r
                     | ~r | abs r | double r | approx r
                     | tail∗(r1, r2, r3) | tail_sq(r1, r2)
assignment lists λ ::= val (x1, ... , xn) = lforce x λ
                     | val (x1, ... , xn) = rforce x λ
                     | val x = susp λ in
                                 ((x1, ... , xn),
                                  (x1, ... , xm))
                               end λ
                     | val x = r λ | empty
sign tests       σ ::= sign r | signtest (r1 ± r2)
                               with λ in σ end
functions        φ ::= fn (x1, ... , xn) =>
                         let λ in σ end
                     | fn (x1, ... , xn) =>
                         let λ in φ end
```

Figure 5: Target language.

The target language of the compilation is presented in Figure 5. It is designed to be easily converted to SML, so its semantics is best explained by referring to SML. In the syntactic category of reals, the symbol $*$ varies over the operations $\{+, -, \times\}$. The values of the syntactic category of reals are translated either into floating-point numbers, or expansions. Each of the operations in the target language has a very definite notion of which of the two types it expects (and floats are considered subtypes of expansions). However, we chose not to make this distinction explicit and did not introduce separate types for floats and expansions in the target language. The reason is that we do not plan to do any programming in this language directly, but rather just use it for intermediate representation of programs before they are converted into SML or C.

The target language operations $\oplus$, $\ominus$ and $\otimes$ are interpreted as corresponding floating-point operations. They expect floating-point input, and produce floating-point output. The exact operations $+$, $-$ and $\times$ are translated into the appropriate exact operations on the data type of expansions. Constants are always floating-point values. The `tail` constructs compute the roundoffs from their corresponding floating-point operation. For example, $\texttt{tail}_*(a, b, a \oplus b)$ will compute the roundoff from the addition $a \oplus b$ following Knuth's theorem. The construct `double` is multiplication by 2 on expansions, and `approx` returns a floating-point number approximating the passed expansion with a relative error of 2ϵ.

To describe the role of `susp`, `lforce` and `rforce` constructs, we need to make a clear distinction between stages and phases of computation in the target language. The source program contains nested functional expression which we refer to as *stages*. Once it is compiled into the target language, every stage gets transformed into a stage of the target language, which consists of four computational *phases* of increased precision. The first phase carries out the computa-

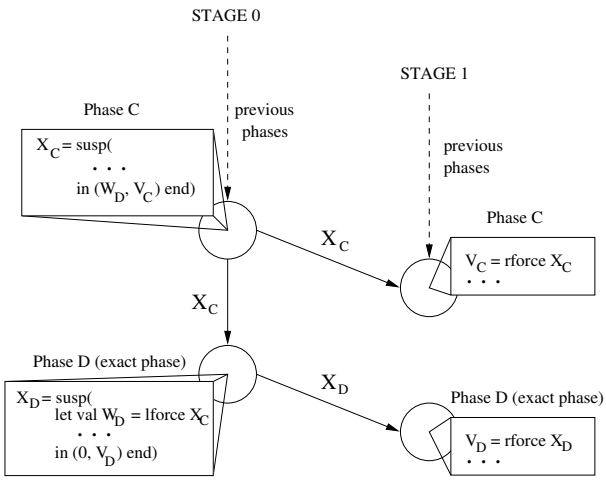

Figure 6: Passing intermediate results between phases and stages using `susp`, `lforce` and `rforce`.

tion in floating-point, and the other phases mix in elements of exact computation as hinted in the previous section. The computation of these other phases have to be *suspended*, since their results are needed only when the floating-point calculations carried out by the first phase of the final stage failed to determine the sign. Thus, the notion of stages refers to *partial evaluation* of code, while the notion of phases refers to *lazy evaluation* of code.

Going back to the target language, `susp` creates a piece of code, a *suspension*, to be evaluated when requested by `rforce` or `lforce`. It gives a mechanism to pass intermediate results between different stages, and between different phases of the same stage. The output from a suspension contains two tuples of intermediate values. The first tuple consists of intermediate values to be passed to some later phase of the current stage, and the second tuple consists of intermediate values intended for the following stage. The first tuple can be recovered by `lforce`-ing the suspension, and the second tuple by `rforce`-ing it (see Figure 6).

The `sign` function returns a sign of an expansion. The construct `signtest` first checks whether the magnitude $|r_1|$ of the tested value is bigger than the magnitude $|r_2|$ of the roundoff error. If so, it returns the sign of r_1. Otherwise, it cannot determine the sign of r_1 with certainty, so it undertakes the computation of the next phase λ, followed by sign test σ. Values r_1 and r_2 are assumed to be floating-point.

4. COMPILATION

To describe the compilation process, first notice that the source program, according to the grammar of the source language (Figure 3), can be viewed as a nonempty sequence of assignment lists, each representing a single stage of computation. Each of these stages is separately compiled into four target phases which are meant to perform the computation of the stage with increasing precision, as described in Section 2. At the end, these pieces of target code are pasted together in a target program, according to specific templates, so that sign checks are performed between subsequent phases, while respecting the staging specified in the source program.

The whole process is formalized using five judgments – four for compiling source stages into their target counterparts, and one judgment to compose all the obtained target stages and phases together into a target program. This section describes the compilation process in more detail, explains some decisions in designing the compilation judgments and illustrates representative rules of the judgments through several examples. Selected subset of rules is presented in the appendix. The complete definition of the judgments can be found in [7].

Before proceeding further, there is one technicality to notice. Namely, it can be assumed, without loss of generality, that the source program to be compiled is in a very specific format. First, we require that all of its assignments consist of a *single* binary operation acting on two other *variables* (rather then on two arbitrary source expressions), or of a single unary operation acting on another variable. The second requirement is that the source program does not contain any floating-point constants – all the constants are replaced by fresh variables for error analysis purposes, and then put back into the target code at the end of the compilation.

It is trivial to transform the source program so that it complies with these two prerequisites, so we do not present the formalization of this procedure. In our implementation it is carried out in parallel with the parsing of the source program.

In the following section, we illustrate the various phases of the process with the compilation of the source expression

```
F  =  fn [a, b] =>
      let val ab = a - b
          val ab2 = sq ab

          fn [c] =>
          let val d = c × ab2
          end
      end
```

4.1 Compiling the stages

The first phase, phase A, of the predicate performs all the source operations approximately in floating-point. The expression compiler determines the appropriate error bounds for the generated code following the equations (2). As can be noticed from these formulas, the relative error is a rational quantity that depends solely on the structure of the source program, while its permanent depends on the input arguments as well, and hence must be computed at runtime. The job of the expression compiler is to determine the relative error of the expression, and insert code into the target program that will compute the permanent and perform checks to see whether the obtained results correctly determine the sign of the source expression.

The rest of this section presents a formalization of the error analysis and program transformation mentioned above. In order to describe the compilation for the phase A, we rely on the judgment

$$E_1 \vdash_A \alpha \rightsquigarrow \lambda; r_1, r_2 / E_2$$

This judgment relates a list α of source assignments to the list λ of corresponding phase A target assignments. Expressions r_1 and r_2 are from the syntactic category of reals in the target language (Figure 5). The expression r_1 is to be tested for sign at the end of the phase, and the expression r_2 is an upper bound on the roundoff error. The assignments

source code	target code	context
phase A		
$\texttt{val ab} = \texttt{a} - \texttt{b}$	$\texttt{val ab}^A = \texttt{a}^A \ominus \texttt{b}^A$	$\texttt{ab}^A : \mathcal{O}_A(\epsilon)$ $\texttt{ab}^P \rhd \texttt{abs}(\texttt{ab}^A)$
$\texttt{val ab2} = \texttt{sq ab}$	$\texttt{val ab2}^A = \texttt{ab}^A \otimes \texttt{ab}^A$	$\texttt{ab2}^A : \mathcal{O}_A(3\epsilon + 3\epsilon^2 + \epsilon^3)$
phase B		
$\texttt{val ab} = \texttt{a} - \texttt{b}$	–	$\texttt{ab}^B : \mathcal{O}_B(\epsilon)$ $\texttt{ab}^B \rhd \texttt{ab}^A$
$\texttt{val ab2} = \texttt{sq ab}$	$\texttt{val ab2}^B = \texttt{sq ab}^A$	$\texttt{ab2}^B : \mathcal{O}_B(2\epsilon + 3\epsilon^2 + \epsilon^3)$
phase C		
$\texttt{val ab} = \texttt{a} - \texttt{b}$	$\texttt{val ab}^C = \texttt{tail}_-(\texttt{a}^A, \texttt{b}^A, \texttt{ab}^A)$	$\texttt{ab}^C : \mathcal{O}_C(0, \epsilon, 0)$
$\texttt{val ab2} = \texttt{sq ab}$	$\texttt{val ab2}^C = \texttt{double}(\texttt{ab}^A \otimes \texttt{ab}^C)$	$\texttt{ab2}^C : \mathcal{O}_C(3\epsilon^2 + 3\epsilon^3, 2\epsilon\frac{1+\epsilon}{1-\epsilon}, \epsilon)$
phase D		
$\texttt{val ab} = \texttt{a} - \texttt{b}$	–	$\texttt{ab}^D \rhd \texttt{ab}^C$
$\texttt{val ab2} = \texttt{sq ab}$	$\texttt{val ab2}^D = \texttt{double}(\texttt{ab}^A \times \texttt{ab}^C) + \texttt{sq}(\texttt{ab}^C)$	–

Table 1: Compilation of the first stage of F.

target code	testing value	error estimate	context
phase A			
$\texttt{val d}^A = \texttt{c}^A \otimes \texttt{ab2}^A$	$\texttt{d}^A$	$\lceil \frac{(1+\epsilon)^2(3\epsilon+3\epsilon^2+\epsilon^3)}{1-\epsilon} \rceil_{fp} \otimes \texttt{abs}(\texttt{d}^A)$	$\texttt{d}^A : \mathcal{O}_A(4\epsilon + 6\epsilon^2 + 4\epsilon^3 + \epsilon^4)$ $\texttt{d}^P \rhd \texttt{abs}(\texttt{d}^A)$
phase B			
$\texttt{val d}^B = \texttt{c}^A \times \texttt{ab2}^B$	$\texttt{approx}(\texttt{d}^B)$	$\lceil \frac{(1+\epsilon)^2(2\epsilon+3\epsilon^2+\epsilon^3)}{1-2\epsilon} \rceil_{fp} \otimes \texttt{abs}(\texttt{d}^A)$	$\texttt{d}^B = \mathcal{O}_B(2\epsilon + 5\epsilon^2 + 4\epsilon^3 + \epsilon^4)$
phase C			
$\texttt{val d}^C = \texttt{c}^A \otimes \texttt{ab2}^C$	$\texttt{d}^C$	$\lceil \frac{5\epsilon^2+12\epsilon^3+6\epsilon^4-4\epsilon^5-3\epsilon^6}{(1-\epsilon)^2} \rceil_{fp} \otimes \texttt{abs}(\texttt{d}^A)$	$\texttt{d}^C : \mathcal{O}_C(\frac{5\epsilon^2+2\epsilon^3-3\epsilon^4}{1-\epsilon}, 2\epsilon(\frac{1+\epsilon}{1-\epsilon})^2, 2\epsilon + \epsilon^2)$
phase D			
$\texttt{val d}^D = \texttt{c}^A \times \texttt{ab2}^D$	$\texttt{d}^B + \texttt{d}^D$	–	

Table 2: Compilation of the second stage of F. The source code for this stage consists of the single assignment $\texttt{val d} = \texttt{c} \times \texttt{ab2}$.

in λ will perform the phase A calculations and compute the appropriate permanent.

The contexts E_1 and E_2 deserve special attention. They are sets relating target language variables with their error estimates. The grammar for their generation is presented below.

$$\begin{array}{ll} \text{contexts} & E ::= \cdot \mid x : \tau, E \mid x \rhd r, E \\ \text{errors} & \tau ::= \mathcal{O}_A(\delta) \mid \mathcal{O}_B(\delta) \mid \mathcal{O}_C(\delta, \iota, \rho) \mid \mathcal{O}_D \mid \mathcal{P} \end{array}$$

Each variable in a context is bound in one of the four phases of the computation (A, B, C or D), and will have error estimates that are appropriate for that phase of the computation ($\mathcal{O}_A(\delta)$, $\mathcal{O}_B(\delta)$, $\mathcal{O}_C(\delta, \iota, \rho)$ and $\mathcal{O}_D$), where δ, ι and ρ are rational numbers (Figure 7). For example, if the error relation $x : \mathcal{O}_A(\delta) \in E$, that means that the variable x which is bound in phase A, has been estimated by the compiler to have a relative error *bounded* from above by the rational number δ. Similar meaning can be ascribed to the error assignments $x : \mathcal{O}_B(\delta)$ for phase B. Phase C, on the other hand, is a mix of approximate and exact computations, and there are three rational values δ, ι and ρ that govern phase C error estimations. We don't describe their meaning in this paper, as it would take too much space, but selected formulas for their derivation can be found in the appendix and the complete definition is in [7]. Phase D is the exact phase, so there are no error estimates to associate with phase D variables. Finally, the temporary variables introduced to hold parts of the permanent are not analyzed for error. We still place them into the error contexts, just for clarity, but with the error tag $\mathcal{P}$. To reduce clutter, the error estimate of a variable x in a context E will be denoted simply as $E(x)$, as it will always be clear from the rule in which phase the variable is bound. In addition to the error estimates, the contexts contain substitutions of variables by target language real expressions ($x \rhd r$). If some compilation rule needs to emit into target code a variable for which there is a substitution in the context, the substituting expression will be emitted instead. This serves two purposes. First, we can use it to express that certain variables in the code are just placeholders for floating-point constants – a situation occurring, as explained before, because of an assumed stricter form of the source programs. Second, it lets us optimize, in a single pass of the compiler, the code for computing the permanent of the expression - a process that will be illustrated below.

Now that we have lain out the structure of the contexts E_1 and E_2 in the judgment we are defining, we can describe their purpose. Simply, the compilation with the judgment starts with the context E_1, and ends with E_2. So, E_2 is in fact E_1 enlarged with the new variables, error estimates and substitutions introduced during the compilation. The context E_2 is returned so that it can be threaded into other rules.

Going back to the analysis for the expression F, given on the previous page, we illustrate how its phase A can be compiled using the above judgment. First of all, the expression F is specified as two stages: the one executing

$\texttt{val ab} = \texttt{a} - \texttt{b}$ and $\texttt{val ab2} = \texttt{sq ab}$, and the other one executing the assignment $\texttt{val d} = \texttt{c} \times \texttt{ab2}$. Each of the stages will be compiled into four phases of assignments. The compilation for phase A starts by breaking down each stage of the source program into individual assignments. The rule is the following.

$$\frac{E_1 \vdash_A \texttt{val x = } e \rightsquigarrow \lambda_H; s_1, s_2 / E' \quad E' \vdash_A \alpha \rightsquigarrow \lambda_T; r_1, r_2 / E_2}{E_1 \vdash_A \texttt{val x = } e\ \alpha \rightsquigarrow \lambda_H \lambda_T; r_1, r_2 / E_2}$$

The rule "folds" the functionality of the compiler across the list of source assignments, carrying the context from one assignment to the next. Notice that the expressions s_1 and s_2 are never used – only the last expression in the assignment list is ever tested for sign. Now, to compile the assignment $\texttt{val ab} = \texttt{a} - \texttt{b}$, we need a rule applicable to subtraction of input arguments. Input arguments are assumed to be errorless, so the following rule applies.

$$\frac{E(x_1^A) = E(x_2^A) = 0}{E \quad \vdash_A \quad \begin{array}{l}\texttt{val } y \texttt{ = } x_1 \texttt{ - } x_2 \rightsquigarrow \texttt{val } y^A \texttt{ = } x_1^A \ominus x_2^A; \\ y^A, 0 / E, y^A : \mathcal{O}_A(\epsilon), y^P : \mathcal{P}, y^P \rhd \texttt{abs}(y^A)\end{array}}$$

When applied to the assignment $\texttt{val ab} = \texttt{a} - \texttt{b}$, the meta variables y, x_1 and x_2 are instantiated to ab, a and b respectively. The rule then emits the target code for the assignment to $\texttt{ab}^A$ (the superscripts A indicate that the target variable is bound in the phase A of the predicate). For the purpose of bookkeeping, the rule must also extend the context with information about the relative error and the permanent of $\texttt{ab}^A$. The relative error of $\texttt{ab}^A$ is ϵ, so the rule generates the error estimate $\texttt{ab}^A : \mathcal{O}_A(\epsilon)$. Finally, the permanent $\texttt{ab}^P$ of $\texttt{ab}^A$ is equal to $|\texttt{ab}^A|$, and the substitution context is extended with $\texttt{ab}^P \rhd \texttt{abs}(\texttt{ab}^A)$.

Next in the assignment list from our example is the assignment $\texttt{val ab2} = \texttt{sq ab}$. The squaring operation is handled by the rule

$$\frac{x_1^P \rhd x_1^A \text{ or } x_1^P \rhd \texttt{abs}(x_1^A) \in E}{E \quad \vdash_A \quad \begin{array}{l}\texttt{val } y \texttt{ = sq}(x_1) \rightsquigarrow \texttt{val } y^A \texttt{ = } x_1^A \otimes x_1^A; \\ y^A, \lceil \frac{(1+\epsilon)^2(2\delta_1+\delta_1^2)}{1-\epsilon} \rceil_{fp} \otimes y^A / \\ E, y^A : \mathcal{O}_A(\epsilon + (1+\epsilon)(2\delta_1 + \delta_1^2)), \\ y^P : \mathcal{P}, y^P \rhd y^A\end{array}}$$

In the assignment to ab2, the meta variables x_1 and y of this rule are instantiated to ab and ab2 respectively. Then the meta variable x_1^P becomes $\texttt{ab}^P$. But $\texttt{ab}^P$ has already been introduced into the context with a substitution $\texttt{ab}^P \rhd \texttt{abs}(\texttt{ab}^A)$. Thus, the premises of this rule are satisfied, and it can be applied. The meta variable δ_1 from the rule refers to the relative error of the variable x_1^A as read from the context, i.e. $\mathcal{O}_A(\delta_1) = E(x_1^A)$. In our example of assignment to ab2, the variable x_1^A is instantiated to $\texttt{ab}^A$ and δ_1 is instantiated to ϵ. The produced permanent for $\texttt{ab2}^A$ is $\texttt{ab2}^A$ itself, explaining why we avoided emitting any code for permanent computation so far. Any separate computation of the permanent for ab2 would have been just a waste of effort, since it is already computed by the main thread of the filter. For future use, however, this rule stores the substitution $\texttt{ab2}^P \rhd \texttt{ab2}^A$ into the context. The relative error for ab2 is computed as $\epsilon + (1+\epsilon)(2\delta_1 + \delta_1^2) = (3\epsilon + 3\epsilon^2 + \epsilon^3)$ and is stored into the context.

That finishes the compilation of phase A of the first stage. The next stage contains the assignment $\texttt{val d} = \texttt{c} \times \texttt{ab2}$, and its phase A target code is obtained by the rule

$$\frac{E(x_1^A) = 0 \qquad x_2^P \rhd x_2^A \text{ or } x_2^P \rhd \texttt{abs}(x_2^A) \in E}{E \quad \vdash_A \quad \begin{array}{l}\texttt{val } y \texttt{ = } x_1 \times x_2 \rightsquigarrow \texttt{val } y^A \texttt{ = } x_1^A \otimes x_2^A; \\ y^A, \lceil \frac{(1+\epsilon)^2 \delta_2}{1-\epsilon} \rceil_{fp} \otimes \texttt{abs}(y^A) / \\ E, y^A : \mathcal{O}_A(\epsilon + (1+\epsilon)\delta_2), y^P : \mathcal{P}, \\ \quad y^P \rhd \texttt{abs}(y^A)\end{array}}$$

The rule compiles the source assignment into $\texttt{val d}^A = \texttt{c}^A \otimes \texttt{ab2}^A$ and expands the current context with the error estimate $\texttt{d}^A : \mathcal{O}_A(4\epsilon + 6\epsilon^2 + 4\epsilon^3 + \epsilon^4)$ and the substitution $\texttt{d}^P \rhd \texttt{abs}(d^A)$.

The remaining phases for F are obtained in a similar way, and the reader is refered to the Appendix for a selected set of rules for the described judgments. The complete definition of the rules can be found in [7]. The steps in the derivation for the two stages, including the changes in the judgment contexts, are presented in Table 1 and Table 2 respectively. In addition to the target code and the contexts, Table 2 also shows, for each of the phases, the testing value and error estimate (recall that only the testing value and the error estimate of the last stage are actually emitted into the target code). As can be seen from Table 2, the testing values for the four phases of the second stage are $\texttt{d}^A$, $\texttt{approx}(\texttt{d}^B)$, $\texttt{d}^C$ and $\texttt{d}^B + \texttt{d}^D$, respectively. In the first three phases, these will be checked against the rounding errors to determine if they have the correct sign. In phase D, the testing value is actually the exact value of the expression. The error estimates for the second stage are obtained from the corresponding rounding errors using (3), producing a quick floating-point test for the sign of the testing value. The error estimates are represented in the table in a symbolic form. It is important to notice that all of them are known in compile time, and are emitted into target code as floating-point constants[1]. So, for example, $\lceil \frac{(1+\epsilon)^2(3\epsilon_3\epsilon^2+\epsilon^3)}{1-\epsilon} \rceil_{fp} = \texttt{3.33067e-16}$ and $\lceil \frac{(1+\epsilon)^2(2\epsilon+3\epsilon^2+\epsilon^3)}{1-2\epsilon} \rceil_{fp} = \texttt{2.22045e-16}$.

4.2 Compiling the program

Once all the stages of the source program have been compiled, they need to be pasted together into a target program, but in such a way that the phases can "communicate" their intermediate results. For illustration, the target code resulting from the compilation of the expression F is presented in Figure 7. The translation is done through a new judgment

$$E_1 \vdash_P \pi; x_B, x_C, x_D \rightsquigarrow \varphi / V_B, V_C, V_D$$

which takes a source program π and compiles it into a target program φ. This judgment works in a bottom-up manner – the later stages are pasted in first (recall that a source program is a "list" of stages; the judgment first processes the tail of the list, and then pastes in the head stages). Thus, it is possible that the target program φ will not have all of its variables bound – some of them might have been introduced in one of the previous stages, and thus will be compiled and bound by the judgment only later. The meta variables x_B, x_C and x_D hold object-code variables, freshly allocated in the previous stage to hold that stage's suspensions, and then passed to the current stage to be rforce'd if needed. The

[1] In the actual SML and C implementations, these values are calculated in an initialization routine, rather than placed in the code as decimal constants, in order to avoid rounding errors in the decimal-to-binary conversion.

```
fn [a^A, b^A] =>
let val ab^A = a^A ⊖ b^A
    val ab2^A = ab^A ⊗ ab^A
    val y_B = susp
                val ab2^B = sq ab^A
              in ((), ab2) end
    val y_C = susp
                val ab^C = tail_(a^A, b^A, ab^A)
                val ab2^C = double(ab^A ⊗ ab^C)
              in ((ab^C), (ab2^C)) end
    val y_D = susp
                val (ab^C) = lforce y_C
                val ab2^D = double(ab^A × ab^C)
                          + sq ab^C
              in ((), ab2^D) end
in
    fn [c^A] =>
    let val d^A = c^A ⊗ ab2^A
    in
        signtest (d^A ± (3.33067e−16 ⊗ abs(d^A)))
        with
           val (ab2^B) = rforce y_B
           val d^B = c^A × ab2^B
           val y^BX = approx(d^B)
        in signtest
            (y^BX ± (2.22045e−16 ⊗ abs(d^A)))
           with
              val (ab2^C) = rforce (y_C)
              val d^C = c^A ⊗ ab2^C
           in signtest
               (d^C ± (2.22045e−16 ⊗ y^BX ⊕
                       6.16298e−32 ⊗ abs(d^A)))
              with
                 val (ab2^D) = rforce y_D
                 val d^D = c^A × ab2^D
              in sign(d^B + d^D) end
           end
        end
    end
end
```

Figure 7: Target code for the example expression F.

variables V_B, V_C and V_D hold the object-code variables that the mentioned suspensions should populate with intermediate values. They are passed back to the previous stage so that the stage can be correctly constructed.

To determine which object-code variables will be passed via suspensions to a particular phase, we use the following function.

$$\mathrm{fv}(\lambda, S) = (S \cup \text{free variables of } \lambda) \setminus \text{bound variables of } \lambda$$

For example, if λ_D is the assignment list for the exact phase (phase D) of the *last* stage in a program π, its free variables will be $V_D = \mathrm{fv}(\lambda_D, \emptyset)$. Some of these free variables will be bound in the λ_A, λ_B or λ_C list of the same stage, but some will have to be passed by a suspension from the exact phase of the previous stage (see Figure 6). The variables to be placed in this suspension are therefore all characterized by the fact that they are introduced in the exact phase of some previous stage. Thus, their set is $V_D \cap \mathrm{dom}_D\, E$, where $\mathrm{dom}_D\, E$ is the set of variables from the context E that have phase D error estimates.

We can similarly determine the suspensions for the phase C of the last stage. Since phase C needs to bind some of the variables from λ_D, we don't just consider the free variables of λ_C, but rather set $V_C = \mathrm{fv}(\lambda_C, V_D)$. As before, some of these variables will be bound in λ_A and λ_B, but those that are not will need to be passed via suspension from the phase C of the preceding stage. These variables are in the set $V_C \cap \mathrm{dom}_C\, E$, where $\mathrm{dom}_C\, E$ is, analogously to the phase D case, the set of object-code variables from context E bound in some of the previous C phases.

In a similar way, the phase B will request the set $V_B \cap \mathrm{dom}_B\, E$ where $V_B = \mathrm{fv}(\lambda_B, V_C)$ passed as a suspension from phase B of the preceding stage. Finally, phase A doesn't require any variable passing, since the computations of this phase are always carried out immediately in each stage, and are never suspended.

The above discussion motivates the following rule of the $\vdash_P$ judgment. The rule applies only if π is a single-stage program, and since the judgment is recursively applied, it serves to compile the *last* stage of the source program.

$$\frac{\begin{array}{l} E, x_i^A : \mathcal{O}_A(0) \vdash_A \alpha \rightsquigarrow \lambda_A;\ r_1^A, r_2^A \big/ E_1 \\ \qquad E_1 \vdash_B \alpha \rightsquigarrow \lambda_B;\ r_1^B, r_2^B \big/ E_2 \\ \qquad E_2 \vdash_C \alpha \rightsquigarrow \lambda_C;\ r_1^C, r_2^C \big/ E_3 \\ \qquad E_3 \vdash_D \alpha \rightsquigarrow \lambda_D;\ r_1^D \big/ E_4 \end{array}}{\begin{array}{l} E \vdash_P \texttt{fn [}x_1, \ldots, x_n\texttt{] => let } \alpha \texttt{ end}; x_B, x_C, x_D \\ \quad \rightsquigarrow \Phi \,\big/ V_B \cap \mathrm{dom}_B\, E, V_C \cap \mathrm{dom}_C\, E, V_D \cap \mathrm{dom}_D\, E \end{array}}$$

where Φ is defined as

```
fn [x_1,...,x_n] =>
let λ_A
in signtest (r_1^A ± r_2^A) with
      val (V_B ∩ dom_B E) = rforce(x_B)
      λ_B
      val y^BX = r_1^B
   in signtest (y^BX ± r_2^B) with
         val (V_C ∩ dom_C E) = rforce(x_C)
         λ_C
      in signtest (r_1^C ± ⌈2ε(1+ε)^2/(1−ε)⌉_fp ⊗ y^BX ⊕ r_2^C)
         with
            val (V_D ∩ dom_D E) = rforce(x_D)
            λ_D
         in sign (r_1^D) end
      end
   end
end
```

Similar analysis of variable passing can be performed if the stage considered is not the last one. Then one only needs to take into account that some object-code variables might be requested from the subsequent stage, and factor them in when creating the suspensions. The rule that handles this case is

$$\frac{\begin{array}{l} E, x_i^A : O(0) \vdash_A \alpha \rightsquigarrow \lambda_A; r_1^A, r_2^A \big/ E_1 \\ \qquad E_1 \vdash_B \alpha \rightsquigarrow \lambda_B; r_1^B, r_2^B \big/ E_2 \\ \qquad E_2 \vdash_C \alpha \rightsquigarrow \lambda_C; r_1^C, r_2^C \big/ E_3 \\ \qquad E_3 \vdash_D \alpha \rightsquigarrow \lambda_D; r_1^D \big/ E_4 \\ \qquad E_4 \vdash_P \pi; y_B, y_C, y_D \rightsquigarrow \varphi \big/ U_B, U_C, U_D \end{array}}{\begin{array}{l} E \vdash_P \texttt{fn [}x_1, \ldots, x_n\texttt{] => let } \alpha\ \pi \texttt{ end}; x_B, x_C, x_D \\ \quad \rightsquigarrow \Phi \big/ V_B \cap \mathrm{dom}_B\, E, V_C \cap \mathrm{dom}_C\, E, V_D \cap \mathrm{dom}_D\, E \end{array}}$$

where $V_D = \text{fv}(\lambda_D, U_D)$, $V_C = \text{fv}(\lambda_C, U_C \cup V_D)$, $V_B = \text{fv}(\lambda_B, U_B \cup V_C)$, and the program Φ is defined as follows.

```
fn [x_1, ..., x_n] =>
let λ_A
    val y_B = susp
                  val (V_B ∩ dom_B E) = rforce x_B
                  λ_B
              in (V_D ∩ dom_B E, U_B) end
    val y_C = susp
                  val (V_C ∩ dom_C E) = rforce x_C
                  λ_C
              in (V_D ∩ dom_C E, U_C) end
    val y_D = susp
                  val (V_D ∩ dom_D E) = rforce x_D
                  val (V_D ∩ dom_B E) = lforce y_B
                  val (V_D ∩ dom_C E) = lforce y_C
                  λ_D
              in ((), U_D) end
in
    φ
end
```

Finally, if π is a source program, then as described before, it can be assumed that all its assignment expressions consist of a single operation acting only on *variables*, and that its constants c_i are replaced by free variables y_i. The target program φ for π is obtained through the judgment after all these new variables are placed into context with relative error 0 together with their substitutions with constants.

$$E, y_i^A : \mathcal{O}_A(0), y_i^A \triangleright c_i \vdash_P \pi; x_B, x_C, x_D \rightsquigarrow \varphi / V_B, V_C, V_D$$

Notice how the pieces of target code shown in Tables 1 and 2, which represent various stages and phases of computation, are pasted together into the target program from Figure 7. For clarity, the empty suspensions and forcings have been deleted from this target program.

5. PERFORMANCE

We have already mentioned that our automatically generated code for 2- and 3-dimensional Orient, InCircle and InSphere predicates to a large extent resembles that of Shewchuk [11]. Of course, this similarity is hard to quantify, if for no other reason than because our predicates are generated in our target language, while Shewchuk's predicates are in C. Nevertheless, we wanted to measure the extent to which the logical and mathematical differences in the code influence the efficiency of our predicates. For that purpose we translated (automatically) the generated predicates from target language into C and compared the translations against Shewchuk's C implementations. The first test consisted of running the compared predicates on a common set of input entries. Each set had 1000 entries, and each entry was a list of point coordinates, in cardinality and dimension appropriate for the particular predicate. The coordinates of the points were drawn with a uniform random distribution from the set of floating-point numbers with exponents between -63 and 63. The summary of the results is represented in Table 3. As can be seen, our C predicates are of comparable speed with Shewchuk's, except in the case of InSphere where Shewchuk's hand-tuned version is about 2.4 times faster. The InSphere predicate is the most complex of all and it is only natural that it can benefit the most from optimizations.

	Shewchuk's version	Automatically generated version	Ratio
Orient2D	0.208 ms	0.249 ms	1.197
Orient3D	0.707 ms	0.772 ms	1.092
InCircle	6.440 ms	5.600 ms	0.870
InSphere	16.430 ms	39.220 ms	2.387

Table 3: Performance comparison with Shewchuk's predicates. The presented results are times for an average run of a predicate on random inputs.

	Shewchuk's version	Automatically generated version	Ratio
random	1187.1 ms	1410.3 ms	1.19
tilted grid	2060.4 ms	3677.5 ms	1.78
co-circular	1190.2 ms	1578.3 ms	1.33

Table 4: Performance comparison with Shewchuk's predicates for 2d divide-and-conquer Delaunay triangulation.

In particular, one of the most visible differences between our InSphere predicate and Shewchuk's is the number of variables declared in the program. Our version of InSphere declares a new `double` array (which can be of considerable size) for every local variable in the target code intended to hold an exact value of an expansion type. However, a lot of this memory can actually be reused, because only a minor portion of the exact values needs to be accessible throughout the run of the program. This will improve the cache management of the automatically generated programs and certainly increase their time efficiency. However, it is important to notice that this problem is not inherent to the automatically generated predicates, but is due to the naive translation from our target language into C. A better translator could probably decrease these differences considerably.

For the second test we modified *Triangle*, Shewchuk's 2d Delaunay triangulator [10] to use automatically generated predicates. The testing included triangulations of three different sets of 50,000 sample points: uniformly random *in* a unit square, tilted grid and uniformly random *on* a unit circle. The summary of the results is represented in Table 4. As can be seen, our predicates are a bit slower in the degenerate cases of tilted grid and co-circular points. Triangulation of such point-distributions often requires the higher phases of the filter, which are better optimized in Shewchuk's hand-tuned versions.

All the results are obtained on a Pentium II on 266 MHz and 96 MB of RAM.

6. FUTURE WORK

The most immediate extensions of the compiler should focus on exploiting the paradigm of staging even better. Staging of expressions prevents recomputing already obtained intermediate results. However, each stage in the source program translates into four phases of the target program, with four approximations of different precision to the given intermediate result. If a computation ever carries out its phase D it will obtain the exact value of this intermediate result, and could potentially use it to increase the accuracy of the approximations from the inexact phases. It would be interesting and useful to devise a scheme that would exploit

both the adaptive precision arithmetic and the staging in this broader manner.

A longer term goal could be to exploit the structure of the computation to obtain better error bounds. Priest has derived sufficient conditions which guarantee that the result from a certain floating-point operation will actually be computed exactly, i.e. will not incur any roundoff error [8]. While putting this idea in practice will likely require a non-trivial amount of theorem proving, it might still be feasible, since geometric predicates are typically short expressions, and that the time for their compilation is not really crucial.

Finally, one may wonder how to extend the source language with the standard programming constructs such as products, coproducts and functions. Adding functions for the sake of structuring the code will most likely require that every single intermediate variable in the program be replaced with a tuple containing that variable phase A value and a suspension for the other three phases. This is required since now functions in the language can test signs of arbitrary values, even those produced by other functions, so the values have to be equipped with means to compute themselves exactly. But this is likely to be too slow, defeating the whole purpose of the expression compiler. On the other hand, adding recursive functions is even less realistic. Performing error analysis for recursive functions is hard – it is one of the main goals of the whole mathematical field of numerical analysis. Therefore, it seems to be more useful to just add coproducts, since products lose much of their purpose if functions are not around.

7. CONCLUSION

This paper has presented an expression compiler for automatic generation of functions for testing the sign of a given arithmetic expression over *floating-point* constants and variables. In addition to the basic operations of addition, subtraction, multiplication, squaring and negation, our expressions can contain anonymous functions and thus exploit the optimization technique of staging, that is well-known in functional programming. The output of the compiler is a target program in a suitably designed intermediate language, which can be easily converted to SML or, in case of single-stage programs, to C.

Our method is an extension to arbitrary expressions of the idea of Shewchuk [11], which he employed to develop quick robust predicates for the Orient and InSphere geometric test. In particular, when applied to source expressions for these geometric predicates, our compiler generates code that, to a large extent, resembles that of Shewchuk. The idea behind the approach is to split the computation into several phases of increasing precision (but decreasing speed), each of which builds upon the result of the previous phase, while using forward error analysis to achieve reliable sign tests.

There remain, however, two caveats when generating predicates with this general approach – the produced code works correctly (1) only if no overflow or underflow happen, and (2) only in round-to-nearest, tie-to-even floating-point arithmetic complying with the IEEE standard.

If overflow or underflow happens in the course of the run of some predicate, the expansions holding exact intermediate results may lose bits of information and distort the final outcome. Thus, we need to recognize such situations and, in those supposedly rare cases, rerun the computation in another form of exact arithmetic (say in infinite precision rational numbers). Unfortunately, even though the IEEE standard prescribes flags that can be read to check for overflow and underflow, the Standard Basis Library of ML does not provide any functions for their testing.

As concerning the second requirement, the IEEE standard is implemented on most modern processors. Unfortunately, on the Intel x86 family this is not a default setup. This family uses internal floating-point registers that are larger than 64-bits reserved for values of floating-point type. This property can occasionally make them round incorrectly in the to-nearest mode (for an example, see [8] page 103) and thus destroys the soundness of the language semantics. This default can be changed by setting a processor flag, but again, the Standard Basis Library does not provide any means for it.

We believe that these two described insufficiencies can easily be remedied, and should be if SML is to become a language with serious applications in numerical analysis and scientific computing.

8. REFERENCES

[1] M. O. Benouamer, P. Jaillon, D. Michelucci, and J. M. Moreau. A lazy exact arithmetic. In E. E. Swartzlander, M. J. Irwin, and J. Jullien, editors, *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, pages 242–249, Windsor, Canada, June 1993. IEEE Computer Society Press, Los Alamitos, CA.

[2] S. Fortune and C. J. V. Wyk. Efficient exact arithmetic for computational geometry. In *Ninth Annual Symposium on Computational Geometry*, pages 163–172. Association for Computing Machinery, May 1993.

[3] S. Funke and K. Mehlhorn. LOOK – a lazy object-oriented kernel for geometric computation. In *Proceedings of the 16th Symposium on Computational Geometry*, pages 156–165. ACM, June 2000.

[4] IEEE. IEEE standard for binary floating-point arithmetic. *ACM SIGPLAN Notices*, 22(2):9–25, Feb. 1985.

[5] K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[6] D. Michelucci and J. M. Moreau. Lazy arithmetic. *IEEE Transactions on Computers*, 46(9), September 1997.

[7] A. Nanevski, G. Blelloch, and R. Harper. Automatic generation of staged geometric predicates. Technical Report CMU-CS-01-141, School of Computer Science, Carnegie Mellon University, June 2001.

[8] D. M. Priest. *On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate Computations*. PhD thesis, University of California at Berkeley, Berkeley, California, November 1992.

[9] S. A. Seshia, G. E. Blelloch, and R. W. Harper. A performance comparison of interval arithmetic and error analysis in geometric predicates. Technical Report CMU-CS-00-172, School of Computer Science, Carnegie Mellon University, December 2000.

[10] J. R. Shewchuk. `http://www.cs.cmu.edu/~quake/triangle.html`.

[11] J. R. Shewchuk. *Delaunay Refinement Mesh Generation*. PhD thesis, Carnegie Mellon University, 1997.

APPENDIX

A. COMPILATION RULES

The expression compiling is governed by five judgments. Four of them correspond to the four phases of adaptive computation. They take lists of source language assignment in context and

produce lists of target language assignment. They also return a target floating-point expression (an expression in the syntactic category of reals) to be tested for sign and a target floating-point expression representing the upper bound on the relative error (or a part of it in the case of phase C). The fifth judgment compiles the whole program by putting together all the pieces of target code obtained by the other judgments. It takes a source program and three variables naming suspensions for B, C and D phases, and returns a target program plus lists of variables to be bound in those suspensions, as described Section 4.

In the following text, concatenation of lists of assignments is represented by their juxtaposition. The relative error of a variable x in context is E is refered to as $E(x)$. Only selected rules of each judgement are presented. For the complete definition, the reader is refered to [7].

A.1 First phase

Phase A of the compilation is handled by the judgment $E_1 \vdash_A \alpha \rightsquigarrow \lambda; r_1, r_2 / E_2$. We abbreviate $\delta_1 = E(x_1^A)$ and $\delta_2 = E(x_2^A)$ when the quantities on the right are defined. The rules for the judgment follow below.

$$\frac{E_1 \vdash_A \texttt{val x = } e \rightsquigarrow \lambda_H; s_1, s_2 / E' \quad E' \vdash_A \alpha \rightsquigarrow \lambda_T; r_1, r_2 / E_2}{E_1 \vdash_A \texttt{val x = } e\ \alpha \rightsquigarrow \lambda_H \lambda_T; r_1, r_2 / E_2}$$

A.1.0.1 Addition.

Denote $\mathrm{err}_+^A(\delta_1, \delta_2) = \epsilon + (1+\epsilon)\max(\delta_1, \delta_2)$.

$$\frac{E(x_1^A) = E(x_2^A) = 0}{E \vdash_A \texttt{val } y \texttt{ = } x_1 \texttt{ + } x_2 \rightsquigarrow \texttt{val } y^A \texttt{ = } x_1^A \oplus x_2^A;\ y^A, 0 / E, y^A : \mathcal{O}_A(\epsilon), y^P : \mathcal{P}, y^P \rhd \texttt{abs}(y^A)}$$

$$\frac{E(x_1^A) = 0}{E \vdash_A \texttt{val } y \texttt{ = } x_1 \texttt{ + } x_2 \rightsquigarrow \texttt{val } y^A \texttt{ = } x_1^A \oplus x_2^A \texttt{ val } y^P \texttt{ = abs}(x_1^A) \oplus x_2^P;\ y^A, \lceil \tfrac{1+\epsilon}{1-\epsilon}\delta_2 \rceil_{fp} \otimes y^P / E, y^A : \mathcal{O}_A(\epsilon + \delta_2), y^P : \mathcal{P}}$$

Symmetrically if $E(x_2^A) = 0$.

$$\frac{x_1^P \rhd x_1^A \in E \qquad x_2^P \rhd x_2^A \in E}{E \vdash_A \texttt{val } y \texttt{ = } x_1 \texttt{ + } x_2 \rightsquigarrow \texttt{val } y^A \texttt{ = } x_1^A \oplus x_2^A;\ y^A, \lceil \tfrac{(1+\epsilon)^2}{1-\epsilon} \max(\delta_1, \delta_2) \rceil_{fp} \otimes y^A / E, y^A : \mathcal{O}_A(\mathrm{err}_+^A(\delta_1, \delta_2)), y^P : \mathcal{P}, y^P \rhd y^A}$$

$$E \vdash_A \texttt{val } y \texttt{ = } x_1 \texttt{ + } x_2 \rightsquigarrow \texttt{val } y^A \texttt{ = } x_1^A \oplus x_2^A \texttt{ val } y^P \texttt{ = } x_1^P \oplus x_2^P;\ y^A, \lceil \tfrac{(1+\epsilon)^2}{1-\epsilon} \max(\delta_1, \delta_2) \rceil_{fp} \otimes y^P / E, y^A : \mathcal{O}_A(\mathrm{err}_+^A(\delta_1, \delta_2)), y^P : \mathcal{P}$$

A.1.0.2 Multiplication.

Denote $\mathrm{err}_\times^A(\delta_1, \delta_2) = \epsilon + (1+\epsilon)(\delta_1 + \delta_2 + \delta_1\delta_2)$.

$$\frac{E(x_1^A) = E(x_2^A) = 0}{E \vdash_A \texttt{val } y \texttt{ = } x_1 \times x_2 \rightsquigarrow \texttt{val } y^A \texttt{ = } x_1^A \otimes x_2^A;\ y^A, 0 / E, y^A : \mathcal{O}_A(\epsilon), y^P : \mathcal{P}, y^P \rhd \texttt{abs}(y^A)}$$

$$\frac{E(x_1^A) = 0 \qquad x_2^P \rhd x_2^A \text{ or } x_2^P \rhd \texttt{abs}(x_2^A) \in E}{E \vdash_A \texttt{val } y \texttt{ = } x_1 \times x_2 \rightsquigarrow \texttt{val } y^A \texttt{ = } x_1^A \otimes x_2^A;\ y^A, \lceil \tfrac{(1+\epsilon)^2\delta_2}{1-\epsilon} \rceil_{fp} \otimes \texttt{abs}(y^A) / E, y^A : \mathcal{O}_A(\epsilon + (1+\epsilon)\delta_2), y^P : \mathcal{P}, y^P \rhd \texttt{abs}(y^A)}$$

$$\frac{E(x_1^A) = 0}{E \vdash_A \texttt{val } y \texttt{ = } x_1 \times x_2 \rightsquigarrow \texttt{val } y^A \texttt{ = } x_1^A \otimes x_2^A \texttt{ val } y^P \texttt{ = abs}(x_1^A) \otimes x_2^P;\ y^A, \lceil \tfrac{(1+\epsilon)^2\delta_2}{1-\epsilon} \rceil_{fp} \otimes y^P / E, y^A : \mathcal{O}_A(\epsilon + (1+\epsilon)\delta_2), y^P : \mathcal{P}}$$

Symmetrically if $E(x_2^A) = 0$.

$$\frac{x_1^P \rhd x_1^A \in E \qquad x_2^P \rhd x_2^A \in E}{E \vdash_A \texttt{val } y \texttt{ = } x_1 \times x_2 \rightsquigarrow \texttt{val } y^A \texttt{ = } x_1^A \otimes x_2^A;\ y^A, \lceil \tfrac{(1+\epsilon)^2(\delta_1+\delta_2+\delta_1\delta_2)}{1-\epsilon} \rceil_{fp} \otimes y^A / E, y^A : \mathcal{O}_A(\mathrm{err}_\times^A(\delta_1, \delta_2)), y^P : \mathcal{P}, y^P \rhd y^A}$$

$$\frac{\begin{array}{l} x_1^P \rhd x_1^A \text{ or } x_1^P \rhd \texttt{abs}(x_1^A) \in E \\ x_2^P \rhd x_2^A \text{ or } x_2^P \rhd \texttt{abs}(x_2^A) \in E \end{array}}{E \vdash_A \texttt{val } y \texttt{ = } x_1 \times x_2 \rightsquigarrow \texttt{val } y^A \texttt{ = } x_1^A \otimes x_2^A;\ y^A, \lceil \tfrac{(1+\epsilon)^2(\delta_1+\delta_2+\delta_1\delta_2)}{1-\epsilon} \rceil_{fp} \otimes y^A / E, y^A : \mathcal{O}_A(\mathrm{err}_\times^A(\delta_1, \delta_2)), y^P : \mathcal{P}, y^P \rhd \texttt{abs}(y^A)}$$

$$E \vdash_A \texttt{val } y \texttt{ = } x_1 \times x_2 \rightsquigarrow \texttt{val } y^A \texttt{ = } x_1^A \otimes x_2^A \texttt{ val } y^P \texttt{ = } x_1^P \otimes x_2^P;\ y^A, \lceil \tfrac{(1+\epsilon)^2(\delta_1+\delta_2+\delta_1\delta_2)}{1-\epsilon} \rceil_{fp} \otimes y^P / E, y^A : \mathcal{O}_A(\mathrm{err}_\times^A(\delta_1, \delta_2)), y^P : \mathcal{P}$$

A.2 Second phase

The judgment handling phase B is $E_1 \vdash_B \alpha \rightsquigarrow \lambda; r_1, r_2 / E_2$. As before, we denote $\delta_1 = E(x_1^B)$ and $\delta_2 = E(x_2^B)$.

$$\frac{E_1 \vdash_B \texttt{val x = } e \rightsquigarrow \lambda_H; s_1, s_2 / E' \quad E' \vdash_B \alpha \rightsquigarrow \lambda_T; r_1, r_2 / E_2}{E_1 \vdash_B \texttt{val x = } e\ \alpha \rightsquigarrow \lambda_H \lambda_T; r_1, r_2 / E_2}$$

A.2.0.3 Addition.

Denote $\mathrm{err}_+^B(\delta_1, \delta_2) = (1+\epsilon)\max(\delta_1, \delta_2)$.

$$\frac{E(x_1^A) = E(x_2^A) = 0}{E \vdash_B \texttt{val } y \texttt{ = } x_1 \texttt{ + } x_2 \rightsquigarrow \texttt{empty};\ 0, 0 / E, y^B : \mathcal{O}_B(\epsilon), y^B \rhd y^A}$$

$$\frac{E(x_1^A) = 0}{E \vdash_B \texttt{val } y \texttt{ = } x_1 \texttt{ + } x_2 \rightsquigarrow \texttt{val } y^B \texttt{ = } x_1^A \texttt{ + } x_2^B;\ \texttt{approx}(y^B), \left\lceil \tfrac{1+\epsilon}{1-2\epsilon}\delta_2 \right\rceil_{fp} \otimes x_2^P / E, y^B : \mathcal{O}_B(\delta_2)}$$

Similarly if $E(x_2^A) = 0$.

$$E \vdash_B \texttt{val } y \texttt{ = } x_1 \texttt{ + } x_2 \rightsquigarrow \texttt{val } y^B \texttt{ = } x_1^B \texttt{ + } x_2^B;\ \texttt{approx}(y^B), \left\lceil \tfrac{1+\epsilon}{1-2\epsilon} \mathrm{err}_+^B(\delta_1, \delta_2) \right\rceil_{fp} \otimes y^P / E, y^B : \mathcal{O}_B(\mathrm{err}_+^B(\delta_1, \delta_2))$$

A.2.0.4 Multiplication.

Denote $\mathrm{err}_\times^B(\delta_1, \delta_2) = (1+\epsilon)(\delta_1 + \delta_2 + \delta_1\delta_2)$.

$$\frac{E(x_1^A) = E(x_2^A) = 0}{E \vdash_B \texttt{val } y \texttt{ = } x_1 \times x_2 \rightsquigarrow \texttt{empty};\ 0, 0 / E, y^B : \mathcal{O}_B(\epsilon), y^B \rhd y^A}$$

$$\frac{E(x_1^A) = 0}{E \vdash_B \texttt{val } y \texttt{ = } x_1 \times x_2 \rightsquigarrow \texttt{val } y^B \texttt{ = } x_1^A \times x_2^B;\ \texttt{approx}(y^B), \left\lceil \tfrac{(1+\epsilon)^2}{1-2\epsilon}\delta_2 \right\rceil_{fp} \otimes y^P / E, y^B : \mathcal{O}_B((1+\epsilon)\delta_2)}$$

Similarly if $E(x_2^A) = 0$.

$$E \quad \vdash_B \quad \texttt{val } y \texttt{ = } x_1 \times x_2 \rightsquigarrow \texttt{val } y^B \texttt{ = } x_1^B \times x_2^B; \\ \texttt{approx}(y^B), \left\lceil \tfrac{1+\epsilon}{1-2\epsilon} \mathrm{err}_\times^B(\delta_1, \delta_2) \right\rceil_{fp} \otimes y^P \big/ \\ E, y^B : \mathcal{O}_B(\mathrm{err}_\times^B(\delta_1, \delta_2))$$

A.3 Third phase

The judgment for phase C is $E_1 \vdash_B \alpha \rightsquigarrow \lambda; r_1, r_2 / E_2$. The expression r_2 is now just one summand in the bound on the absolute error. See the definition of the judgment $\vdash_P$ for its use in the target program. Notational abbreviation for this section are $\Delta_1 = (\delta_1, \iota_1, \rho_1) = E(x_1^C)$ and $\Delta_2 = (\delta_2, \iota_2, \rho_2) = E(x_2^C)$ when the context E contains variables x_1^C and x_2^C.

$$\frac{\begin{array}{lll} E_1 & \vdash_C & \texttt{val x = } e \rightsquigarrow \lambda_H; s_1, s_2 / E' \\ E' & \vdash_C & \alpha \rightsquigarrow \lambda_T; r_1, r_2 / E_2 \end{array}}{\begin{array}{lll} E_1 & \vdash_C & \texttt{val x = } e\ \alpha \rightsquigarrow \lambda_H \lambda_T; r_1, r_2 / E_2 \end{array}}$$

A.3.0.5 Addition.

To simplify the presentation, we introduce the following notation.

$$\begin{aligned} &\mathrm{err}_+^C((\delta_1, \iota_1, \rho_1), (\delta_2, \iota_2, \rho_2)) \\ &\quad = \quad (\delta_+^C, \frac{1+\epsilon}{1-\epsilon} \max(\iota_1, \iota_2), \mathrm{err}_+^A(\rho_1, \rho_2)) \end{aligned}$$

where

$$\delta_+^C = (1+\epsilon)(\epsilon \max(\iota_1, \iota_2) + \max(\delta_1, \delta_2))$$

$$\frac{E(x_1^A) = E(x_2^A) = 0}{\begin{array}{lll} E & \vdash_C & \texttt{val } y \texttt{ = } x_1 \texttt{ + } x_2 \rightsquigarrow \\ & & \texttt{val } y^C \texttt{ = tail}_+(x_1^A, x_2^A, y^A);\ 0, 0 / \\ & & E, y^C : \mathcal{O}_C(0, \epsilon, 0) \end{array}}$$

$$\frac{E(x_1^A) = 0}{\begin{array}{lll} E & \vdash_C & \texttt{val } y \texttt{ = } x_1 \texttt{ + } x_2 \rightsquigarrow \texttt{ empty;} \\ & & x_2^C, \left\lceil \tfrac{(1+\epsilon)^2}{1-\epsilon} \delta_2 \right\rceil_{fp} \otimes x_2^P \big/ \\ & & E, y^C : \mathcal{O}_C(\Delta_2), y^C \rhd x_2^C \end{array}}$$

Similarly for $E(x_2^A) = 0$.

$$E \quad \vdash_C \quad \texttt{val } y \texttt{ = } x_1 \texttt{ + } x_2 \rightsquigarrow \texttt{val } y^C \texttt{ = } x_1^C \oplus x_2^C; \\ y^C, \left\lceil \tfrac{(1+\epsilon)^2}{1-\epsilon} \delta_+^C \right\rceil_{fp} \otimes y^P \big/ \\ E, y^C : \mathcal{O}_C(\mathrm{err}_+^C(\Delta_1, \Delta_2))$$

A.3.0.6 Multiplication.

Here, the error functions are as follows.

$$\mathrm{err}_\times^{C0}(\delta, \iota, \rho) = (\epsilon\iota + \delta, \frac{1+\epsilon}{1-\epsilon}\iota, \epsilon + (1+\epsilon)\rho)$$

$$\begin{aligned} &\mathrm{err}_\times^C((\delta_1, \iota_1, \rho_1), (\delta_2, \iota_2, \rho_2)) \\ &\quad = \quad (\delta_\times^C, \frac{1+\epsilon}{1-2\epsilon-\epsilon^2}(\iota_1 + \iota_2), \mathrm{err}_\times^A(\rho_1, \rho_2)) \end{aligned}$$

where

$$\begin{aligned} \delta_\times^C \quad = \quad & \Big[(2\epsilon + \epsilon^2)(\iota_1 + \iota_2) + \\ & + (\rho_1\iota_2 + \iota_1\rho_2) + \delta_1(1 + \iota_2 + \rho_2) + \\ & + \delta_2(1 + \iota_1 + \rho_1) + \iota_1\iota_2 + \delta_1\delta_2 \Big](1+\epsilon) \end{aligned}$$

$$\frac{E(x_1^A) = E(x_2^A) = 0}{\begin{array}{lll} E & \vdash_C & \texttt{val } y \texttt{ = } x_1 \times x_2 \rightsquigarrow \\ & & \texttt{val } y^C \texttt{ = tail}_\times(x_1^A, x_2^A, y^A)\ \lambda;\ 0, 0 / \\ & & E, y^C : \mathcal{O}_C(0, \epsilon, 0) \end{array}}$$

$$\frac{E(x_1^A) = 0}{\begin{array}{lll} E & \vdash_C & \texttt{val } y \texttt{ = } x_1 \times x_2 \rightsquigarrow \texttt{val } y^C \texttt{ = } x_1^A \otimes x_2^C; \\ & & y^C, \left\lceil \tfrac{(1+\epsilon)^2}{1-\epsilon}(\epsilon\iota_2 + \delta_2) \right\rceil_{fp} \otimes y^P \big/ \\ & & E, y^C : \mathcal{O}_C(\mathrm{err}_\times^{C0}(\Delta_2)) \end{array}}$$

Similarly for $E(x_2^A) = 0$.

$$E \quad \vdash_C \quad \texttt{val } y \texttt{ = } x_1 \times x_2 \rightsquigarrow \\ \texttt{val } y^C \texttt{ = } (x_1^A \otimes x_2^C) \oplus (x_1^C \otimes x_2^A); \\ y^C, \left\lceil \tfrac{(1+\epsilon)^2}{1-\epsilon} \delta_\times^C \right\rceil_{fp} \otimes y^P \big/ \\ E, y^C : \mathcal{O}_C(\mathrm{err}_\times^C(\Delta_1, \Delta_2))$$

A.4 Fourth phase

The phase D of the filter is exact, so there is no need for error functions or estimates in the judgment. Thus, the judgment has the form $E_1 \vdash_D \alpha \rightsquigarrow \lambda; r / E_2$, and is defined below.

$$\frac{E_1 \vdash_D \texttt{val x = } e \rightsquigarrow \lambda_H; s / E' \qquad E' \vdash_D \alpha \rightsquigarrow \lambda_T; r / E_2}{\begin{array}{lll} E_1 & \vdash_D & \texttt{val x = } e\ \alpha \rightsquigarrow \lambda_H \lambda_T; r / E_2 \end{array}}$$

A.4.0.7 Addition.

$$\frac{E(x_1^A) = E(x_2^A) = 0}{\begin{array}{lll} E & \vdash_D & \texttt{val } y \texttt{ = } x_1 \texttt{ + } x_2\ \alpha \rightsquigarrow \texttt{ empty;}\ 0 / \\ & & E, y^D : \mathcal{O}_D, y^D \rhd y^C \end{array}}$$

$$\frac{E(x_1^A) = 0}{\begin{array}{lll} E & \vdash_D & \texttt{val } y \texttt{ = } x_1 \texttt{ + } x_2 \rightsquigarrow \texttt{ empty;}\ y^B + x_2^D / \\ & & E, y^D : \mathcal{O}_D, y^D \rhd x_2^D \end{array}}$$

Similarly if $E(x_2^A) = 0$.

$$E \quad \vdash_D \quad \texttt{val } y \texttt{ = } x_1 \texttt{ + } x_2 \rightsquigarrow \\ \texttt{val } y^D \texttt{ = } x_1^D \texttt{ + } x_2^D;\ y^B + y^D / E, y^D : \mathcal{O}_D$$

A.4.0.8 Multiplication.

$$\frac{E(x_1^A) = E(x_2^A) = 0}{\begin{array}{lll} E & \vdash_D & \texttt{val } y \texttt{ = } x_1 \times x_2 \rightsquigarrow \texttt{ empty;}\ 0 / \\ & & E, y^D : \mathcal{O}_D, y^D \rhd y^C \end{array}}$$

$$\frac{E(x_1^A) = 0}{\begin{array}{lll} E & \vdash_D & \texttt{val } y \texttt{ = } x_1 \times x_2 \rightsquigarrow \\ & & \texttt{val } y^D \texttt{ = } x_1^A \times x_2^D;\ y^B + y^D / E, y^D : \mathcal{O}_D \end{array}}$$

Similarly if $E(x_2^A) = 0$.

$$E \quad \vdash_D \quad \texttt{val } y \texttt{ = } x_1 \times x_2 \rightsquigarrow \\ \texttt{val } y^D \texttt{ = (}x_1^B \times x_2^D\texttt{)+(}x_1^D \times x_2^B\texttt{)+(}x_1^D \times x_2^D\texttt{);} \\ y^B + y^D / E, y^D : \mathcal{O}_D$$