모든 컴퓨터 과학자가 알아야할 부동 소수점의 모든 것

What Every Computer Scientist Should Know About Floating-Point Arithmetic

Jaebum Lee

$\mathcal{JB}$

이 문서의 모든 내용은 `http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html` 에 공개 되어 있는 **모든 컴퓨터 과학자가 알아야 하는 부동 소수점 연산의 모든 것(What Every Computer Scientist Should Know About Floating-Point Arithmetic)** 을 번역한 것입니다. 양질의 무료 컴퓨터 관련 문서를 한국어로 공급하기 위한 프로젝트의 일환으로, 이 문서는 `http://itguru.tistory.com` 에서 무료로 배포되고 있습니다. (오픈북 프로젝트에 관해 자세한 내용은 `http://itguru.tistory.com` 을 참조하시기 바랍니다) 문서 전체 번역은 이재범(`kev0960@gmail.com`) 외 수 많은 사람들이 수정에 도움을 주셨습니다.

원 문서에서의 주석은 모두 각주(footnote) 로 달았고, 특별한 설명이 더 필요하다고 생각된 부분은 역자 주를 책 여백(marginnote)에 달았습니다. 또한, 마지막의 참고 문헌 목록은 생략하였으니, 혹시 참고 문헌을 찾고 싶은 분들은 원문을 참조하시기 바랍니다.

부동 소수점 연산(Floating-point arithmetic)은 소수의 사람들만이 다루는 주제로 여겨져왔습니다. 이는 많은 컴퓨터 시스템 상에서 부동 소수점 연산이 사용된다는 사실을 생각해보았을 때 매우 놀랄 만한 일인데요, 거의 모든 컴퓨터 언어에서는 부동 소수점 데이터 타입이 있고, PC 를 비롯한 슈퍼 컴퓨터들에 까지 거의 모든 장치에는 부동 소수점 가속기 (floating-point accelerator) 들이 있습니다. 많은 컴파일러들은 부동 소수점 알고리즘을 컴파일 하는데 사용될 것이며, 사실상 모든 운영체제에서는 오버플로우 (overflow) 와 같은 부동 소수점 예외(exception)를 처리할 수 있어야만 합니다. 이 문서에서는 부동 소수점의 여러 특징 중에서 컴퓨터 시스템의 설계에 가장 직접적으로 영향을 주었던 부분에 대해 다룰 것입니다. 일단, 그 시작으로, 부동 소수점 표기법의 설립 배경과, 반올림 오차, 그리고 IEEE 부동 소수점 표준에 대해 다룰 것이며, 끝 부분에서는 어떻게 컴퓨터 제작자들이 부동 소수점을 더 잘 지원할 수 있는지에 대해 설명하도록 할 것입니다.

중요 키워드 : 비정규화 수(Denormalized number), 예외(Exception), 부동 소수점(Floating-point), 부동 소수점 표준(Floating-point standard), 점진적 언더플로우(Gradual underflow), 보호 숫자(Guard digit), NaN, 오버플로우(Overflow), 상대 오차(Relative error), 반올림 오차(Rounding error), 반올림 모드(Rounding mode), ulp, 언더 플로우(Underflow)

차 례

들어가며

컴퓨터 시스템 제작자들은 종종 부동 소수점 연산에 대한 지식을 요구로 합니다. 하지만, 놀랍게도 부동 소수점에 대해 자세하게 설명한 것들은 극히 드물지요. 심지어 부동 소수점에 대해 다루는 몇 안되는 책들 중 하나인 Pat Sterbenz 의 *Floating-Point Computation* 은 절판된지 오래입니다. 이 문서는 부동 소수점 연산의 여러 특징 중에서 컴퓨터 시스템 설계에 직접적으로 영향을 주는 것들에 대해 다루고자 합니다. 이를 위해 크게 3 개의 부분으로 문서를 나누었는데, 첫 번째 반올림 오차에선 부동 소수점 덧셈, 뺄셈, 곱셈, 나눗셈시 어떠한 방식으로 반올림을 하느냐에 따라 달라지는 양상을 살펴볼 것입니다. 또한, 반올림 오차, ulps, 그리고 상대 오차를 계산하는 두 가지 방식을 다룰 것입니다. 두 번째 IEEE 표준 에선 IEEE 부동 소수점 표준에 대해 이야기 하는데, 이는 상업 하드웨어 업체들에게 빠르게 채택되고 있는 방식 중 하나 입니다. 세번째 시스템적 측면 에서는 부동 소수점을 컴퓨터 시스템 설계 측면에서 살펴볼 것인데, 예를 들어 명령어 세트(instruction set) 설계, 컴파일러 최적화, 예외 처리 등에 대해 다룰 것입니다.

참고로, 이 문서에서 저는 간단한 산수로 쉽게 보일 수 있는 것들에 대해서는 증명하지 않고 넘어가도록 하였습니다. 특히 모든 (단순한) 문장에 대해 시시 콜콜하게 집고 넘어가는 것은 불필요하다고 생각되기 때문이지요. 글의 논지에서 벗어나는 자질구레한 증명들은 뒤에 자세한 것들을 참고하시기 바랍니다. 참고로 이 장에서 등장하는 여러 정리들의 증명 뒤에는 □ 로 표시되어 있는데, 증명을 생략했을 경우 그냥 정리만 써 놓았습니다.

제 1 장

반올림 오차 (Rounding Error)

무한히 많은 실수들을 유한개의 비트로 표현하기 위해서는 근사적으로 표현해야 합니다. 비록 정수 개수도 무한하다 해도, 대부분의 프로그램에서 사용하는 스케일의 정수들은 32 비트 이내로 저장할 수 있습니다. 반면에, 실수 간의 연산에서는 32 비트 이내로 계산 결과를 정확하게 표현할 수 있는 경우가 드뭅니다. 따라서, 부동 소수점 연산의 결과는 반드시 반올림을 통해 유한 비트 표현 방식으로 나타내져야만 합니다. 그렇기 때문에 반올림 오차는 부동 소수점 연산에서의 어쩔수 없는 특징이기도 하지요. 상대 오차와 Ulp 에서는 어떻게 반올림 오차를 측정하는지에 대해 설명할 것입니다.

어차피 대부분의 부동 소수점 연산에서 반올림 오차가 발생하므로, 산술 연산 작업에서 필요 이상 보다 살짝 더 발생하는 오차에 대해 관심을 가져야 할까요? 보호 숫자 (Guard Digits)에서는 두 인접한 수를 뺄 때 발생하는 오차를 줄이기 위한 방법으로 도입된 보호 숫자라는 개념에 대해 설명할 것입니다. 보호 숫자 라는 개념은 IBM 에서 중요하게 생각해온 개념으로 1968년에 IBM 의 System/360 아키텍쳐의 배정밀도 (double precision) 형식에 추가된 이후로 모든 장치들에게 도입되었습니다. 이 보호 숫자의 효용성에 대해 설명하기 위해 2 가지 예제를 살펴볼 것입니다.

IEEE 표준에서는 단순히 보호 숫자를 필수적으로 사용해야 한다는 것 보다 더 많은 것을 요구합니다. IEEE 표준에서는 덧셈, 뺄셈, 곱셈, 나눗셈, 그리고 제곱근 연산에 대한 알고리즘을 제공하며, 이 표준에 따라 만들어진 장치에서는 연산시 제공된 알고리즘을 사용한 결과와 같은 결과가 나오도록 요구합니다. 따라서, 어떤 프로그램이 다른 장치로 옮겨졌을 때, 만일 그 장치가 IEEE 표준을 지원하는 장치라면, 기본 연산 결과는 장치에 관계없이 모두 같을 것입니다. 따라서 IEEE 표준을 지원하는 장치 끼리 매우 쉽게 프로그램을 포팅 (porting) 할 수 있게 되지요. 더 자세한 사항은 정확히 반올림되는 연산들 섹션에서 살펴볼 것입니다.

1.1 부동소수점 형식 (Floating-point Formats)

그 동안 실수를 표현하는 방식으로 꽤 많은 종류의 형태가 제안되었지만, 현재까지 가장 많이 사용하는 방식은 부동 소수점 표현입니다. [1] 부동 소수점 방식은 (항상

부동 소수점 이란, 말 그대로 소수점의 위치가 고정된 것이 아닌, 단어 그래로 둥둥 떠다니며 위치가 바뀐다는 의미 입니다

짝수라고 가정되는) 밑수(base) β 와 정밀도(precision) p 를 인자로 가지며 표현하게 됩니다. 예를 들어서 $\beta = 10$ 이고 $p = 3$ 이라면, 0.1 을 표현했을 때 1.00×10^{-1} 이렇게 표기할 수 있게 됩니다. 만일 $\beta = 2$ 이고 $p = 24$ 라면, 0.1 은 정확하게 표현할 수 없고, 단순히 근사적으로 $1.10011001100110011001101 \times 2^{-4}$ 로 표시할 수 밖에 없습니다. 일반적으로, 부동 소수점 수는 $\pm d.dd...d \times \beta^e$ 로 표현되며, $d.dd...d$ 부분의 경우 가수(significand) 라고 불리며 p 자리 수 입니다. 즉, 정확히 말하면 $\pm d_0.d_1d_2....d_{p-1} \times \beta^e$ 로 나타내겠지요.

$$\pm (d_0 + d_1\beta^{-1} + ... + d_{p-1}\beta^{-(p-1)})\beta^e, (0 \leq d_i \leq \beta) \tag{1.1}$$

여기서 **부동 소수점 수**라는 단어는, 식 1.1 의 형태로 정확히 표현되는 수를 지칭하는 말로 사용하도록 하겠습니다. 부동 소수점의 또다른 중요한 인자로 지수의 크기 범위를 나타내는 값들이 있는데, 이를 각각 θ_{max} 와 θ_{min} 이라고 지칭합니다. 따라서 β^p 개의 가능한 가수들과, $\theta_{max} - \theta_{min} + 1$ 개의 가능한 지수들을 표현하기 하기 위해서는 총

$$[\log_2(e_{max} - e_{min} + 1)] + [\log_2(\beta^p)] + 1$$

비트가 필요로 하며, 끝에 붙은 $+1$ 은 부호 비트를 위해 필요한 추가적인 비트를 나타냅니다.

모든 실수가 부동 소수점 형식으로 표현될 수 없는 이유는 크게 두 가지가 있는데, 가장 흔한 이유로, 0.1 을 예로 들을 수 있습니다. 비록 0.1 은 십진수로 유한개의 자리수로 표현할 수 있지만, 이진법으로 표현시에는 무한개의 자리수가 필요하게 됩니다. 따라서 $\beta = 2$ 일 경우, 0.1 은 두 개의 부동 소수점 수 사이에 위치하게 되지만, 그 들 둘 다 0.1 은 아닙니다. 다른 경우로는 부동 소수점으로 표현 가능한 범위를 벗어난 경우인데, 만일 절대값이 $\beta \times \beta^{e_{max}}$ 보다 크거나, $1.0 \times \beta^{e_{min}}$ 보다 작을 경우에 부동 소수점으로 표현할 수 없게 됩니다. 이 문서에서는 후자 보다는 전자로 인해 표현될 수 없는 실수들에 대해 다룰 것이지만, 후자의 경우에 대해서는 무한대 와 정규화 되지 않는 수 을 참고하시기 바랍니다.

부동 소수점 표현은 반드시 유일할 필요는 없습니다. 예를 들어서 0.01×10^1 과 1.00×10^{-1} 은 모두 0.1 을 표현하지요. 만일 맨 앞자리가 0 이 아니라면 (즉, 식 1.1 에서 $d_0 \neq 0$) 부동 소수점 표현이 **정규화(normalized)** 되었다고 말합니다. 따라서, 1.00×10^{-1} 는 정규화 되있는 것이고, 0.01×10^1 는 정규화 되어 있지 않은 것입니다. $\beta = 2, p = 3, \theta_{min} = -1, \theta_{max} = 2$ 인 경우, 그림 1.1 에서 나타난 것 처럼 16개의 정규화된 부동 소수점이 가능합니다. 하지만, 이렇게 정규화된 부동 소수점들 만을 사용하기로 제한하게 된다면, 0 을 표현하지 못한다는 문제점이 생기게 됩니다. 이러한 문제를 해결하기 위한 가장 자연스러운 방법은, $1.0 \times \beta^{e_{min}}$ 로 표현하는 것인 데, 이러한

1) 그 외의 제안된 방식으로는 floating slash 와 signed logarithm 등의 표현 방식이 있습니다. [Matula and Kornerup 1985; Swartzlander and Alexopoulos 1975].

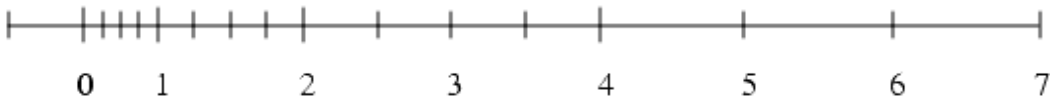

그림 1.1: $\beta = 2, p = 3, e_{min} = -1, e_{max} = 2$ 일 때의 정규화 된 수들

방법을 사용할 경우 만일 지수를 k 비트로 표현한다면 총 $2^k - 1$ 만큼의 수 만을 지수로 사용할 수 있게 됩니다 (왜냐하면 나머지 하나는 0 을 위해 남겨놓았으므로)

참고로 부동 소수점에서의 $\times$ 는 부동 소수점 표기 중 일부지, 실제로 부동 소수점 수의 곱셈 연산을 의미하는 것이 아닙니다. 예를 들어서 $(2.5 \times 10^{-3}) \times (4.0 \times 10^2)$ 에서는 오직 한 번의 부동 소수점 곱셈이 발생하게 됩니다.

1.2 상대 오차와 Ulp

부동 소수점 연산시에 반올림 오차는 필연적으로 발생하는 것이기 때문에 그 오차를 측정하는 것은 매우 중요한 일입니다. 일단 여기서 기본적으로 $\beta = 10, p = 3$ 인 부동 소수점들을 고려하도록 합시다. 만일 부동 소수점 연산의 결과가 3.12×10^{-2} 이고, 정확한 결과는 .0314 였다면, 마지막 자리에서 2 의 오차가 있었음을 알 수 있습니다. 마찬가지로 만일 정확한 값은 .0314159 인데, 부동 소수점 표현으로 3.14×10^{-2} 로 나타났다면, 마지막 자리에서 .159 의 오차가 발생하였음을 알 수 있습니다. 따라서 일반적으로, 만일 실수 z 를 부동 소수점 수 $d.d...d \times \beta^e$ 로 근사시켰다면, 마지막 자리에서 발생하는 오차는 $|d.d...d - (z/\beta^e)|\beta^{p-1}$ 임을 알 수 있습니다. [2] [3] Ulps 는 *Units in the last place* 의 약자로 **마지막 자리의 단위** 를 의미합니다. 부동 소수점 연산의 결과가 참값과 최대한으로 가깝다 하더라도, 그 오차는 최대 .5ulp 까지 날 수 있습니다. 부동 소수점 수와 참값과의 차이를 측정하는 다른 방식으로 **상대 오차(relative error)** 가 있는데, 이는 단순히 두 수의 차이를 참값으로 나눈 것입니다. 예를 들어, 3.14159 를 3.14×10^0 으로 근사하였다면, 상대 오차의 크기는 $.00159/3.14159 \approx .0005$ 가 됩니다.

.5ulp 에 해당하는 오차의 크기를 계산하기 위해서, 만일 어떤 실수가 부동소수점 수 $d.dd...dd \times \beta^e$ 로 표현되었다고 하면, .5ulp 의 크기는 $0.00...00\beta' \times \beta^e$ 이 됩니다. 여기서 β' 은 $\beta/2$ 입니다. 부동 소수점 수의 가수가 p 자리라면, 그 오차에는 p 개의 0 이 오게 됩니다. 따라서, .5ulp 에 해당하는 오차는 $((\beta/2)\beta^{-p}) \times \beta^e$ 가 됩니다. $d.dd...dd \times \beta^e$ 꼴의 모든 숫자들이 같은 크기의 절대 오차(absolute error) 을 가지게 되지만, 상대 오차의 경우 $((\beta/2)\beta^{-p}) \times \beta^e/\beta^e$ 와 $((\beta/2)\beta^{-p}) \times \beta^e/\beta^{e+1}$ 의 범위를 가지게 됩니다.

2) 여기서 z 가 $\beta^{e_{max}} + 1$ 보다 크거나 $\beta^{e_{min}}$ 보다 작은 경우는 고려하지 않습니다. 물론, 추후에 따로 명시하기 전까지 이러한 범위에 있는 수들은 고려하지 않습니다.

3) z' 을 z 를 부동 소수점 수로 근사한 것이라 가정하면,$|d.d...d - (z/\beta^e)|\beta^{p-1}$ 는 $|z' - z|/uls(z')$ 이라 한 것과 같습니다. 참고로 오차를 계산하는 좀더 정확한 공식은 $|z' - z|/uls(z)$ 입니다 - Ed.

이 말은,

$$\frac{1}{2}\beta^{-p} \leq \frac{1}{2}ulp \leq \frac{\beta}{2}\beta^{-p} \tag{1.2}$$

워블(Wobble) 의 사전적 정의은 불안정함, 떨림, 동요 등 입니다.

따라서 .5ulp 에 대응하는 상대 오차의 경우, β 의 배수로 값이 바뀔 수 있으며, 이 인자를 **워블(wobble)** 이라 부릅니다. 식 에서의 상한을 $\epsilon = (\beta/2)\beta^{-p}$ 로 정의하면, 언제나 어떤 실수를 가장 가까운 부동 소수점 수로 반올림 하였을 때 상대 오차는 항상 ϵ 이하며, 이를 **머신 입실론(machine epsilon)** 이라 부릅니다.

위 예제에서 상대 오차는 $.00159/3.14159 \approx .0005$ 였는데, 이와 같이 상대 오차의 크기는 매우 작으므로, 보통 ϵ 의 배수로 많이 나타냅니다. 이 경우 $\epsilon = (\beta/2)\beta^{-p} = 5(10)^{-3} = .005$ 이므로, 상대 오차의 경우 $(.00159/3.14159)/.005)\epsilon \approx 0.1\epsilon$ 이 됩니다.

상대오차와 ulps 의 차이를 설명하기 위해 실수 $x = 12.35$ 를 생각합시다. 이는 $\bar{x} = 1.24 \times 10^1$ 로 근사되며, 오차는 0.5ulps 이고, 상대 오차는 0.8ϵ 입니다. 다음에, 만일 $8\bar{x}$ 을 계산한다면, 참값은 $8x = 98.8$ 이지만, 계산된 값은 $8\bar{x} = 9.92 \times 10^1$ 입니다. 이 경우, 오차는 4.0ulps 가 되었지만, 상대 오차는 0.8ϵ 으로 그대로 입니다. ulps 로 측정된 오차의 크기는 8 배나 커졌지만, 상대오차는 0.8ϵ 로 그대로 이지요. 일반적으로, 밑수가 β 일 때, 상대오차를 일정하게 유지시킨다면 ulps 로 측정된 오차는 β 의 배수로 늘어날 수 있지만, 역으로 식 에서도 나타나듯이, ulps 오차를 고정시킨다면 상대 오차는 β 의 배수로 늘어날 수 있습니다.

반올림으로 발생한 오차를 측정하는 가장 자연스러운 방법은 ulps 로 측정하는 것입니다. 예를 들어, 가장 가까운 부동 소수점 수로 반올림 할 때 발생하는 오차는 .5ulp 보다 작거나 같겠지요. 하지만, 여러가지 연산들이 포함되어 있는 식에서의 반올림 오차를 분석할 때, 상대 오차를 측정하는 것이 좀 더 나은 방법입니다. 반올림 오차에서, 여러 식들을 연산할 때의 상대 오차를 계산하는 방법을 살펴볼 것입니다. ϵ 이 가장 가까운 부동 소수점 수로 반올림 하는 효과를 β 의 배수로 뻥튀기 할 수 있기 때문에 작은 β 를 사용하는 것이 오차를 측정하는데 좀 더 정확합니다.

만일 반올림 오차의 차수(order) 에만 관심을 가진다면 ulps 와 ϵ 은 단순히 β 배 차이이기 때문에 서로 바꿔가며 사용할 수 있습니다. 예를 들어서 만일 부동 소수점 수가 n ulps 만큼의 오차가 있다면, 이 말은 $log\beta n$ 만큼의 자리를 신뢰할 수 없다는 뜻입니다. 만일, 상대 오차가 $n\epsilon$ 이라면,

$$\text{contaminated digits} \approx \log \beta n \tag{1.3}$$

1.3 보호 숫자(Guard Digits)

두 부동 소수점 수의 차이를 계산하는 한 가지 방법으로, 단순히 두 수의 차이를 계산한 뒤에 가장 가까운 부동 소수점 수로 반올림 하는 방법이 있습니다. 이 방법은 두 수의 차이가 매우 크다면, 비효율적인 방법입니다. 예를 들어 $p = 3, 2.14 \times 10^{12} - 1.25 \times 10^{-5}$ 이라면, 계산 결과

$$x = 2.15 \times 10^{12}$$
$$y = .0000000000000000125 \times 10^{12}$$
$$x - y = 2.1499999999999999875 \times 10^{12}$$

로 반올림 하면, 다시 2.15×10^{12} 가 됩니다. 이 모든 자리를 다 계산하기 보다는, 부동 소수점 하드웨어는 보통 고정된 개수의 자리수에서만 연산을 하게 됩니다. 만일, p 개의 자리수만 보관한다고 생각하고, 차이 계산시 더 작은 수를 오른쪽으로 쉬프트 시킨다고 생각하면 (쉬프트시 뒤 자리수는 버려지게 된다), $2.14 \times 10^{12} - 1.25 \times 10^{-5}$ 는

$$x = 2.15 \times 10^{12}$$
$$y = 0.00 \times 10^{12}$$
$$x - y = 2.15 \times 10^{12}$$

가 됩니다. 이 결과는 차이를 계산한 뒤 반올림 한 것과 정확히 같지요. 이번에는 다른 예제를 살펴봅시다. $10.1 - 9.93$ 의 경우

$$x = 1.01 \times 10^{1}$$
$$y = 0.99 \times 10^{1}$$
$$x - y = .02 \times 10^{1}$$

정확한 값은 .17 이므로, 계산된 값과 그 차이는 무려 30ulps 에 달합니다.

Theorem 1.3.1. 인자가 β 와 p 인 부동 소수점 형식을 사용할 때, p 자리를 사용해서 두 부동 소수점 수의 차이를 계산한다면, 그 결과의 상대 오차는 $\beta - 1$ 까지 커질 수 있다.

나머지 모든 뺄셈에서의 가능한 상대오차가 $\beta - 1$ 보다 작다는 사실은 자명하게 보일 수 있습니다.

Proof. $x - y$ 를 계산할 때 상대오차가 $\beta - 1$ 인 경우는 $x = 1.00...0$ 이고, $y = .\rho\rho...\rho$ 일 때, $\rho = \beta - 1$ 이면 된다. 여기서 y 는 p 자리수 이고, 정확한 차이는 $x - y = \beta^{-p}$ 이다. 하지만, 오직 p 자리 만큼만 사용한다고 하였으므로, y 의 가장 오른쪽 자리수는 떨어져 나가게 된다. 따라서, 계산된 차이는 β^{-p+1} 이고, 오차는 $\beta^{-p} - \beta^{-p+1} = \beta^{-p}(\beta - 1)$ 이고, 상대 오차는 $\beta^{-p}(\beta - 1)/\beta^{-p} = \beta - 1$ 이 되므로 성립한다. □

$\beta = 2$ 일 대, 상대 오차는 결과값과 동일하게 될 수 있으며, $\beta = 10$ 이면, 상대 오차는 결과값에 9 배나 커질 수 있습니다. 다르게 말하면, $\beta = 2$ 일 때, 식 1.3 에 따르면, 신뢰할 수 없는 자리수의 경우 $\log_2(1/\epsilon) = \log_2(2^p) = p$ 라는 말입니다. 다시 말해, 모든 p 자리수가 틀렸다는 이야기 이지요. 만일 추가적으로 1 개 자리수가 이와 같은 상황을 방지하기 위해 더해졌다고 가정 해봅시다. (보호 숫자) 이렇게 되면 뺄셈 시에, 더 작은 수는 $p + 1$ 개 자리수로 분할되며, 뺄셈의 결과는 p 자리수로 반올림 됩니다. 보호 숫자를 도입하게 될 경우, 위 예제는

$$x = 2.15 \times 10^{12}$$
$$y = 0.00 \times 10^{12}$$
$$x - y = 2.15 \times 10^{12}$$

가 되고 결과는 정확합니다. 1 개의 보호 숫자를 통해, 상대 오차는 ϵ 보다 커질 수 있으며, $110 - 8.59$ 에서 나타나는 것 처럼,

$$x = 1.10 \times 10^{2}$$
$$y = .085 \times 10^{2}$$
$$x - y = 1.015 \times 10^{2}$$

이 결과는 102 로 반올림 되며, 정확한 결과인 101.41 에 비교해 볼 때 상대 오차는 .006 이고 이는 $\epsilon = .005$ 보다 크게 됩니다. 대부분의 경우, 계산 결과의 상대 오차는 ϵ 보다 살짝 커질 수 있으며 정확히 말하자면,

Theorem 1.3.2. 만일 인자 β, p 를 사용한 부동 소수점 형식에서, x 와 y 사이의 뺄셈을 $p+1$ 자리의 수 *(1 개의 보호 숫자)* 로 수행하였다면, 그 결과의 상대 오차는 2ϵ 보다 작다.

이에 대한 증명은 반올림 오차에서 할 것입니다. 참고로 위 증명에서 x 와 y 가 양수인지 음수인지 상관이 없기 때문에 뺄셈에 대해 증명한다면 자동으로 덧셈에 대해서 증명이 됩니다.

1.4 지워짐 (Cancellation)

앞선 내용에 따르면 보호 숫자 없이는 매우 가까운 두 수의 뺄셈은 엄청나게 큰 상대 오차를 발생시킨다는 사실을 알 수 있었습니다. 다시 말해, 뺄셈이 포함된 식은 매우 큰 상대 오차를 발생시키게 되는데, 심지어는 너무나 상대 오차가 커져서, 모든 숫자들이 의미 없게 만들어 버릴 수 도 있었습니다. (정리 1.3.1). 두 개의 가까운 수를 뺄 때, 앞 자리 숫자 (most significant digits) 들은 서로 지워지게 됩니다. 이 때 두 가지 방식의 지워짐 (cancellation) 이 가능한데, 이는 각각 나쁜 (catastrophic) 과 착한 (benign) 지워짐이 있습니다.

참고로 숫자가 '의미 없게' 되었다 라는 말은, 영어에서 오염되었다 (contaminated) 라고 합니다.

먼저 나쁜 지워짐 (catastrophic cancellation) 은, 두 피연산자들에 이미 반올림 오차가 적용되어 있을 때 발생하게 됩니다. 예를 들어 2차 방정식의 근의 공식을 계산할 때, $b^2 - 4ac$ 를 계산하게 됩니다. 이 때, b^2 와 $4ac$ 에는 이미 부동 소수점 곱셈을 수행하였기 때문에 반올림 오차가 반영된 상태입니다. 만일 이들이 가장 가까운 부동 소수점으로 반올림 된다고 할 때 (따라서 오차가 .5ulp 이내), 이들 사이에 뺄셈을 한다면, 두 피연산자 차이가 적기 때문에 (이렇다고 앞에서 가정함) 매우 정확한 자리수들이 모두 지워지고, 뒤에 반올림으로 인해 신뢰할 수 없는 자리수들만 남겨놓게 됩니다. 따라서, 그 오차는 기존의 .5ulps 에서 수 ulps 로 뛰어 오르게 됩니다. 예를 들어서 $b = 3.34, a = 1.22, c = 2.28$ 이라고 가정해봅시다. $b^2 - 4ac$ 의 정확한 값은 .0292

이차 방정식의 근의 공식 $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

입니다. 그런데, b^2 은 11.2 로 반올림되고, $4ac$ 는 11.1 로 반올림 되서 계산 결과가 .1 이 나왔다면, 비록 뺄셈에서 $11.2 - 11.1$ 이 정확히 .1 이 되었다고 해도 이는 무려 700 ulps 의 오차가 됩니다.

$0.1 - 0.0292 = 0.0708$ 이고, 0.0292 의 마지막 자리가 0.0001 이므로 700 ulps 가 된다.

착한 지워짐(benign cancellation) 은 두 개의 정확한 수의 뺄셈시에 발생하게 됩니다. 만일 x, y 모두에 반올림 오차가 없다면, 정리 1.3.2 에 의해 보호 숫자를 이용해서 뺄셈 수행시 $x - y$ 는 매우 작은 상대 오차(2ϵ 이하) 를 가지게 됩니다.

나쁜 지워짐의 가능성을 내포하고 있는 공식을 때로는 이를 포함하고 있지 않도록 변형할 수 있습니다. 예를 들어서 아래와 같은 이차 방정식의 근의 공식을 살펴봅시다.

$$r_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, r_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \tag{1.4}$$

만일 $b^2 >> ac$ 라서 $b^2 - 4ac$ 에서 지워짐이 발생하지 않는다면, 단순히

$$\sqrt{b^2 - 4ac} \approx |b|$$

라고 볼 수 있습니다. 하지만 이 경우 덧셈(뺄셈) 에서 나쁜 지워짐이 발생하게 되는데, 이를 피하기 위해서는 식 1.4 의 분자 분모에 $-b - \sqrt{b^2 - 4ac}$ 를 곱해서,

$$r_1 = \frac{2c}{-b - \sqrt{b^2 - 4ac}}, r_2 = \frac{2c}{-b + \sqrt{b^2 - 4ac}} \tag{1.5}$$

를 요도할 수 있습니다. 만일 $b^2 >> ac$ 이고 $b > 0$ 이라면, r_1 을 식 을 이용해서 계산하는 것은 나쁜 지워짐을 발생시키게 됩니다. 하지만 반대로 식 를 이용해서 r_1 을 계산하고 식 을 이용해서 r_2 를 계산하면 이를 막을 수 있습니다. 만일 $b < 0$ 의 경우 반대로 하면 되겠지요.

$x^2 - y^2$ 또한 나쁜 지워짐을 야기시킬 수 있는 식입니다. 각각을 제곱해서 곱하는 것 보다는 $(x - y)(x + y)$ 를 계산하는 것이 더 정확합니다. 이차 방정식 근의 공식과는 달리, 뺄셈에 있음에도 불구하고 그 뺄셈이 착한 뺄셈이기 때문에 큰 반올림 오차 없이 처리할 수 있습니다. 정리 1.3.2 에 의해 $x - y$ 의 상대 오차는 최대 2ϵ 이기 때문이죠. $x + y$ 도 마찬가지 입니다. 두 개의 작은 상대 오차를 가지는 부동 소수점 수를 곱한 그 결과 역시 작은 상대 오차 (섹션 반올림 오차 참고) 를 내므로 효율적이라 볼 수 있습니다.

계산된 결과와 실제 정확한 결과를 구분하기 위해서 다음과 같은 기호를 사용하도록 하겠습니다. 예를 들어서 $x - y$ 를 실제로 정확한 차이 라고 한다면, $x \ominus y$ 는 컴퓨터로 계산된 차이 가 됩니다. (즉, 반올림 오차를 포함한) 비슷하게도, $\oplus, \otimes, \oslash$ 등이 각각 덧셈, 곱셈, 나눗셈을 의미합니다. 또한, 대문자로 쓰여진 함수들, 예를 들어 $LN(x), SQRT(x)$ 는 컴퓨터로 계산된 값을 의미하고, 원래 쓰이던 표현 $(\ln x, \sqrt{x})$ 등은 정확히 계산된 값을 의미합니다.

비록 $(x \ominus y) \otimes (x \oplus y)$ 는 $x^2 - y^2$ 의 훌륭한 근사라고 해도, 만일 x, y 자체가 실제값 $\hat{x}, \hat{y}$ 의 근사값일 수 도 있습니다. 예를 들어 $\hat{x}, \hat{y}$ 가 10진수로 정확히 표현되지만 이진수로는 정확히 표현되지 않는 값일 수 도 있습니다. 이 경우 비록 $x \ominus y$ 는 $x - y$ 의

좋은 근사 일 수 있지만 $\hat{x}-\hat{y}$ 에 비해 큰 상대 오차를 가질 수 있기 때문에 $(x+y)(x-y)$ 의 정확도는 x^2-y^2 에 비해 크게 효과적이지 않을 수 있습니다. 사실 $(x+y)(x-y)$ 나 x^2-y^2 의 경우 계산하는데 걸리는 시간이 거의 비슷하기 때문에, 그나마 좀더 정확한 전자가 선호되겠지만, 일반적인 경우 나쁜 지워짐을 착한 지워짐으로 바꾼다고 해서 근본적인 문제 (입력값 자체가 근사값이라는 사실) 을 해결할 수 없습니다. 하지만, 지워짐 자체를 아예 발생하게 하지 않는다면, 입력되는 데이터가 근사값이라고 해도 매우 의미 있을 것입니다.

x^2-y^2 은 $(x+y)(x-y)$ 로 씀으로써 나쁜 지워짐을 착한 지워짐으로 바꿀 수 있었기 때문에 정확도를 향상시킬 수 있었습니다. 다음 예제로, 나쁜 지워짐을 가지고 있지만 모두 착한 지워짐만으로 바꿀 수 있는 흥미로운 예를 하나 살펴볼 것입니다.

삼각형의 넓이는 아래 식 1.6 에 따라 각 변의 길이 a, b, c 를 통해 표현할 수 있습니다.

$$A = \sqrt{s(s-a)(s-b)(s-c)}, \text{where } s = (a+b+c)/2 \tag{1.6}$$

예를 들어 삼각형이 매우 납작하다고 가정합시다 ($a \approx b+c$). 즉, $s \approx a$ 가 됩니다. 따라서, 식 1.6 에서의 $(s-a)$ 항은 근접한 두 값을 빼게 되 것이고, 심지어 그들 중 하나는 반올림 오차를 포함하고 있을 수 도 있습니다. $a = 9.0, b = c = 4.53$ 인 경우 s 의 정확한 값은 9.03 이고 A 는 2.342... 입니다. s 의 계산된 값은 9.05 로 오직 2 ulps 의 오차가 있지만 A 의 계산된 결과는 3.04 로, 무려 70 ulps 의 오차가 발생하게 됩니다. 아래는 식 1.6 을 재구성 해서 납작한 삼각형에 경우에도 정확히 계산할 수 있도록 수정한 것입니다. [Kahan 1986]

$$A = \frac{\sqrt{(a+(b+c))(c-(a-b))(c+(a-b))(a+(b-c))}}{4}, a \geq b \geq c \tag{1.7}$$

만일 a, b, c 가 $a \geq b \geq c$ 를 만족하지 않더라도, 식 1.7 자체는 순환적이기 때문에 그냥 알파벳 순서만 살짝 바꿔주면 됩니다. 위로 수정된 결과를 이용하면 계산된 값은 2.35 가 나오며 이는 오직 1 ulps 의 오차 입니다.

위 예제만을 보더라도 알겠지만 일반적인 경우에도 식 1.6 보다는 식 1.7 가 훨씬 나은 모습을 보여줍니다.

Theorem 1.4.1. 식 *1.7* 를 사용하였을 때 발생하는 반올림 오차는 최대 11ϵ 이다. *(*단, 뺄셈은 보호 숫자를 이용하였고, $e \leq .005$, 그리고 제곱근의 계산 오차는 *0.5ulp* 이내이다.*)*

$\epsilon < .005$ 라는 조건은 대부분의 부동 소수점 시스템에서 만족하고 있습니다. 예를 들어서 $\beta = 2, p \geq 8$ 나 $\beta = 10, p \geq 3$ 만 되도, $\epsilon < .005$ 를 만족합니다. 정리 1.4.1 에서의 식에서 상대 오차를 고려할 때, 위 식 자체가 부동 소수점 연산을 통해 처리된다는 것을 이해하셔야 합니다. 따라서 우리가 실제로 계산하는 식은

$$SQRT((a \oplus (b \oplus c)) \otimes (c \ominus (a \ominus b)) \otimes (c \oplus (a \ominus b)) \otimes (a \oplus (b \ominus c))) \oslash 4 \qquad (1.8)$$

보통 오차 상한선은 매우 크게 잡히는 것이 일반적인데, 예를 들어 정리 1.4.1 에서는 넉넉하게 11ϵ 정도로 잡았는데, 위 예제에서 계산되었던 2.35는 실제 값 2.34216 에 대해 상대 오차가 0.7ϵ 정도 밖에 안됩니다. 이렇게 오차 상한선을 크게 잡는 이유는 나중에 수치적으로 증명할 때 조금 더 편리하기 때문입니다.

마지막 예로 $x << 1$ 일 때의 $(1+x)^n$ 가 착한 지움을 살펴보도록 수정하는 것을 살펴봅시다. 참고로 이 식은 금융 계산에서 자주 쓰이게 되는데, 예컨대 매일 100 달러를 은행 계좌에 집어 넣은다고 가정하고, 연 이율이 6% 라고 생각합시다. 만일 $n = 365$ 이고, $i = .06$ 이라면 1년이 지난 후에 계좌에 누적된 돈의 양은

$$100^{\frac{(1+i/n)^n - 1}{1/n}}$$

달러가 됩니다. $\beta = 2, p = 24$ 일 때 컴퓨터로 계산해본다면 그 결과 $37615.45 가 나오는데, 실제로 정확한 결과는 $37614.05 로 $1.40 의 오차가 발생하게 됩니다. 이 오차가 발생하는 이유는 매우 간단한데, $1 + i/n$ 은 1 에 .0001643836 을 더하는 것이므로, 너무나 값이 작기 때문에 i/n 의 뒷자리 비트들이 잘려나가게 됩니다. 특히 이 반올림 오차는 $1 + i/n$ 이 n 승이 될 때 더 확대 됩니다.

이 문제많은 $(1+i/n)^n$ 은 $e^{n \ln(1+i/n)}$ 로 다시 쓸 수 있는데, 이번에 문제는 $\ln(1+x)$ 를 작은 x 에 대해 연산하는 것이 문제가 됩니다. 이 근사식을 이용하는 한 가지 방법으로 $ln(1 + x) \approx x$ 임을 생각할 수 있는데, 이 경우, 계산된 식이 $37617.26 이 되어 오히려 기존의 오차였던 $1.40 보다 증가한 $3.21 이 됩니다. 하지만 다행이도 $\ln(1 + x)$ 를 정확하게 계산할 수 있는 방법이 있는데 이는 정리 1.4.2 에 나타나 있습니다. [Hewlett-Packard 1982] 이 공식을 이용하면 $37614.07 로 오직 2 센트 오차만 발생하게 되죠.

Theorem 1.4.2. 만일 $ln(1 + x)$ 를 아래 공식을 이용해 계산하면

$$\ln(1+x) = \begin{cases} x & if\ 1 \oplus x = 1 \\ \frac{x \ln(1+x)}{(1+x)-1} & if\ 1 \oplus x \neq 1 \end{cases}$$

일 때 상대 오차는 최대 5ϵ 이다. *(*단, $0 \leq x < 3/4$ 이고, 뺄셈 시 보호 숫자를 사용하고, $e < 0.1$, ln 을 *0.5ulp* 이내로 계산한다.*)*

사실 이 공식은 위를 만족하는 어떤 x 값에 대해 서도 적용 가능하지만, 실제로 우리가 관심을 가지는 것은 그대로 $\ln(1 + x)$ 를 계산시에 나쁜 지워짐이 발생하는 $x << 1$ 일 경우 입니다. 위 식 자체가 매우 이상하게 보일지라도, 왜 이 식이작동하는지는 매우

간단하게 알 수 있습니다. $\ln(1+x)$ 를

$$x(\frac{ln(1+x)}{x}) = x\mu(x)$$

여기서 문제가 되는 부분은 $\mu(x)$ 를 정확하게 계산할 수 있느냐 인데, 왜냐하면 $\mu(x) = \frac{\ln(1+x)}{x}$ 를 계산할 때 $1+x$ 를 연산 시 큰 반올림 오차가 생기기 때문입니다. 하지만 여기서 중요한 사실은 μ 의 값은 $\ln(1+x) \approx x$ 이기에 거의 일정하다는 사실입니다. 따라서 x 를 살짝 바꾸는 것은 큰 오차를 야기하지 않습니다. 다시 말해, 만일 $\bar{x} \approx x$ 라면, $x\mu(\bar{x})$ 를 계산하는 것은 $x\mu(x) = \ln(1+x)$ 의 좋은 근사가 될 것이라는 것이지요. 그런데, $\bar{x}$ 와 $\bar{x}+1$ 모두 정확하게 계산할 수 있는 $\bar{x}$ 를 찾을 수 있나요? 바로 $\bar{x} = (1 \oplus x) \ominus 1$ 을 사용하는 것입니다. 왜냐하면 $\bar{x}+1$ 은 정확히 $1 \oplus x$ 가 될 것이기 때문이죠.

이 지워짐(Cancellation)에서 다루어 왔던 내용들을 정리해보자면, 차이가 거의 없지만 정확하게 알려진 두 수의 뺄셈 시에 (착한 지움) 보호 숫자를 도입한다면 일정 정확도가 보장된다는 것입니다. 또한, 때로는 공식 자체가 부정확한 결과를 낼 수 있지만, 착한 지움을 이용한다면 같은 공식이라도 다르게 연산함으로써 높은 정확도를 낼 수 있게 됩니다. 하지만 이 역시 뺄셈을 수행시 보호 숫자를 도입한 연산이 보장될 때 이야기 입니다. 사실 보호 숫자를 도입하는데 필요한 비용은 크지 않습니다. 왜냐하면 이는 단순히 1 개 비트 정도를 추가하는 것 뿐이기 때문이죠. 예를 들어서 54비트 배정밀도 누산기(adder) 의 경우 1 개 비트를 추가하는데 전체 비용에 2% 정도 밖에 되지 않습니다. 이 정도 비용으로는 여러분은 공식 1.6 과 같은 삼각형 넓이를 구하는 알고리즘이라던지, $\ln(1+x)$ 을 계산하는 알고리즘을 사용할 수 있게 되는 것입니다. 대부분의 현대 컴퓨터들이 보호 숫자를 사용하는 가운데, 일부 시스템 (Cray System 등) 에서는 사용하지 않고 있습니다.

1.5 정확한 반올림 연산

보호 숫자를 이용한 부동 소수점 연산은 정확하게 계산한 다음에 가장 가까운 부동 소수점 수로 반올림 하는 것 보다는 정확하지 않습니다. 이러한 방식으로 계산하는 것을 **정확히 반올림 되었다(exactly rounded)** 라고 부릅니다 [4] . 정리 1.3.2 이전에 바로 나온 예제가 보여주듯이, 보호 숫자를 이용한 연산은 언제나 정확한 반올림 된 결과를 보여주지는 않습니다. 이전 장에서 보호숫자를 이용하여 여러 알고리즘을 처리하는 것을 살펴보았다면 이번 장에서는 정확한 반올림을 요구로 하는 알고리즘들을 살펴볼 것입니다.

아직까지 반올림에 대한 정확한 정의를 내리지 않았습니다. 사실 반올림 자체는 정확히 반을 가루는 예외적인 경우를 제외하고는 매우 직관적입니다. 더 가까운 쪽으로 올리거나 버리면 되는 것인데, 문제는 그 중간에 수가 위치할 경우 입니다. 12.5 는 12

4) 혹은 *correctly rounded* 라고 불리기도 합니다

로 내림이 되어야 할까요 아니면 13 으로 올림이 되어야 할까요. 어떤 학교에서는 아마 0,1,2,3,4 는 내림을 해야 하고 5,6,7,8,9 는 올림을 해야 한다고 가르쳐야 할 것입니다. 이와 같은 방식은 Digital Equipment Corporation 의 VAX 컴퓨터들이 채택하고 있는 방식이기도 합니다. 반올림을 수행하는 다른 방식으로는 마지막 자리가 5 인 경우, 올림을 하던 내림을 하던 두 가지 가능성이 있기 때문에, 50% 의 경우에 올림을 하고 나머지 50% 의 경우의 내림을 수행하도록 하면 된다는 것입니다. 즉 이 방식에서는 항상 반올림 된 수가 짝수가 되도록 반올림을 하게 결정하면 올림과 내림 모두 공평하게 처리할 수 있게 됩니다.. 즉, 12.5 는 13 이 되는 것이 아니라 12 가 되고, 13.5 는 13 이 아니라 14 가 됩니다. 둘 중 어느 방식이 효과적인지는 Reiser and Knuth [1975] 에서 다루고 있으며, 그 결과 항상 짝수로 반올림 하는 것이 효과적이라 결정났습니다.

Theorem 1.5.1. x, y 를 부동 소수점 수라 하고, $x_0 = x, x_1 = (x_0 \ominus y) \oplus y, ..., x_n = (x_{n-1} \ominus y) \oplus y$ 라 정의합시다. 만일 $\oplus$ 와 $\ominus$ 가 모두 정확한 **짝수** 반올림 연산을 수행된다면, 모든 n 에 대해 $x_n = x$ 이거나, $n \leq 1$ 에 대해 $x_n = x_1$ 을 만족한다. *(*단, $n \geq 1$*)*

위 정리 1.5.1 의 내용을 명확히 하자면, 예를 들어 $\beta = 10, p = 3$ 이고 $x = 1.00, y = -.555$ 라고 가정해봅시다. 반올림 시에, 위 순열은 $x_0 \ominus y = 1.56, x_1 = 1.56 \ominus .555 = 1.01, x_1 \ominus y = 1.01 \oplus .555 = 1.57, ...$ 가 되고, 각 순열 마다 x_n 의 크기는 $x_n = 9.45 (n \leq 845)$ 일 때 까지 .01 씩 늘어나게 됩니다. 반면에 짝수 반올림시에, x_n 은 항상 1.00 이 됩니다. 이 예제를 통해 알 수 있는 점은, 그냥 반올림을 사용한다면 계산 결과는 조금씩 증가하게 되는데 짝수 반올림을 사용하는 경우 그러한 현상이 나타나지 않는다는 점입니다. 따라서 이제 부터 이 문서 전체에서 항상 짝수 반올림만을 사용할 것이며, 그냥 반올림이라 칭할 시에, 짝수 반올림을 의미하는 것입니다.

정확한 반올림을 사용하는 한 가지 경우는 바로 다정밀도(multiple precision) 연산입니다. 다정밀도 연산은 보통 두 가지 방식으로 구현하는데, 크기가 큰 가수를 가지고 있는 부동소수점 수를 사용한다던지 (예를 들어 WORD 의 배열로 보관한다는 등), 다정밀도 부동 소수점 수를 보통의 부동 소수점 수의 배열의 합으로써 나타내면 됩니다. 여기서 다룰 방식은 후자로, 보통의 부동 소수점 들을 배열로 구현하는 것은 고급 언어에서 쉽게 코딩할 수 있기 때문입니다. 다만, 이를 위해서는 정확한 반올림 연산을 필요로 합니다.

여기서 WORD 는 컴퓨터 자료형의 한 종류로 보통 32 비트 (4 바이트) 정수를 의미합니다.

이러한 시스템에서, 곱셈 연산은 다음과 같이 구현됩니다. 예를 들어서 다정밀도 부동 소수점 수 x, y 에 대해서 각각이 $x = x_h + x_l$ 이고 $y = y_h + y_l$ 이라 한다면, 둘의 곱은

$$xy = x_h y_h + x_h y_l + x_l y_h + x_l y_l$$

로 주어집니다. 만일 x, y 의 가수가 p 비트라면, 우변의 각각의 항들 역시 가수의 비트가 p 비트일 것이고, x_l, x_h, y_l, y_h 각각은 $\lfloor p/2 \rfloor$ 비트로 나타낼 수 있게 됩니다. 만일 p 가 짝수라면, 간단히 $x_0.x_1...x_{p/2-1}$ 과 $0.0...0x_{p/2}...x_{p-1}$ 의 합으로 나타낼 수

있겠지요. 하지만, p 가 홀수라면 전자와 같이 간단히 두개로 쪼갤 수 없게 되는데, 하지만 음수를 사용함으로써 비트하나를 늘려서 짝수 비트로 만들 수 있습니다. 예를 들어서 만일 $\beta = 2, p = 5, x = .10111$ 일 때 x 는 $x_h = .11$ 와 $x_l = -.00001$ 로 쪼갤 수 있습니다. 수를 쪼개는 방식은 이 방식 말고도 여러 방법이 있는데, Dekker [1971] 덕분에 매우 간단히 계산할 수 있게 되었습니다. 다만, 이 방법으로는 1 개 이상의 보호 숫자가 필요합니다.

Theorem 1.5.2. p 를 부동 소수점 정밀도 *(*단 $\beta > 2$ 일 때, p 는 항상 짝수*)*, 그리고 부동 소수점 연산 시에 정확히 반올림 된다고 가정합시다. 만일 $k = [p/2]$ 라 정의하고, $m = \beta^k + 1$ 이라 정의하면, x 는 $x = x_h + x_l$ 로 분할 할 수 있으며, 이 때

$$x_h = (m \otimes x) \ominus (m \otimes x \ominus x), x_l = x \ominus x_h$$

이고 각각의 x_i 는 $[p/2]$ 비트의 정밀도로 표현 가능하다.

이 정리 1.5.2 가 어떻게 작동하는지 확인하기 위해 $\beta = 10, p = 4, b = 3.476, a = 3.463, c = 3.479$ 라 해봅시다. $b^2 - ac$ 를 가장 가까운 부동 소수점 수로 반올림 하면 .03480 이 되는데, 이 때 $b \otimes b = 12.08, a \otimes c = 12.05$ 이므로 따라서 계산된 $b^2 - ac$ 는 .03 입니다. 이는 무려 오차가 480ulps 나 되지요. 정리 1.5.2 을 이용하면, $b = 3.5 - .024, a = 3.5 - .037, c = 3.5 - .021$ 로 나타낼 수 잇습니다. 이 변형된 형태를 이용하여 계산을 수행해보면, $b^2 = 12.25 - .168 + .000576$ 가 되고 (이 시점에서는 아직 그 값이 계산된 상태는 아님), $ac = 3.5^2 - (3.5 \times .037 + 3.5 \times .021) + .037 \times .021 = 12.25 - .2030 + .00077$ 가 됩니다. 이제, $b^2 - ac$ 를 각각의 항에 대한 뺄셈을 수행한다면 그 결과 $0 \oplus .0350 \ominus .000201 = .03480$ 이 되며 이는 정확히 반올림 된 결과와 같게 됩니다.

정리 1.5.2 이 정확한 반올림을 필요로 한다는 것을 보여주기 위해서 한 가지 예로 $p = 3, \beta = 2, x = 7$ 일 때를 생각해봅시다. $m = 5, mx = 35, m \otimes x = 32$ 가 되는데, 만일 한 개의 보호 숫자로 뺄셈을 수행하게 된다면 $(m \otimes x) \ominus x = 28$ 이 됩니다. 따라서 $x_h = 4, x_l = 3$ 이 되어 x_l 은 $[p/2] = 1$ 비트로 표현할 수 없게 되죠.

이 기법은 후의 증명에서 많이 사용됩니다. 증명시 '크기를 조정하여' 라는 말은 모두 β 의 멱수를 곱해서 가수는 유지시키고, 지수만 바꾼다는 의미 입니다. 많은 경우 지수는 크게 관련이 없기 때문에 이 기법을 사용하여 원하는 범위 내로 주어진 수를 위치 시킬 수 있습니다

정확한 반올림의 마지막 예를 위해, m 을 10 으로 나누는 경우를 생각해봅시다. 그 결과는 실제로 $m/10$ 한 결과와 부동 소수점 수와 일반적으로 다를 것입니다. 하지만 놀랍게도 $\beta = 2$ 인 경우에, $m/10$ 한 결과에 10 을 곱한다면 다시 m 으로 돌아오게 됩니다. 사실 더 일반적인 경우에도 이는 성립하는데 (Kahan 이 이를 멋지게 보였습니다) 증명은 기발하지만 궁금하지 않은 독자들은 IEEE 표준 으로 넘어가시기 바랍니다.

Theorem 1.5.3. $\beta = 2$ 일 때, m, n 이 모두 정수이고, $|m| < 2^{p-1}$ 이고 $n = 2^i + 2^j$ 의 특별한 꼴일 때, 연산시 정확한 반올림을 한다면 $(m \oslash n) \otimes n = m$ 을 만족한다.

Proof. 2의 멱수를 곱하는 것은 아무런 문제가 되지 않는데 왜냐하면 이는 가수를 바꾸는 것이 아니라 단순히 지수만 바꾸는 것이기 때문입니다. 따라서 만일 $q = m/n$ 이라면, 2 의 멱수를 곱해서 $2^{p-1} \leq n < 2^p$ 가 되고, m 역시 마찬가지로 해서 $1/2 < q < 1$ 가

되도록 조정할 수 있게 됩니다. 따라서 $2^{p-2} < m < 2^p$ 가 되죠. 이 때, m 이 p 개의 비트를 가지게 되므로, 소수점 오른쪽으로 최대 한 개의 비트만 가질 수 있게 됩니다. m 의 부호를 바꾸는 것은 증명에 영향을 주지 않으므로 $q > 0$ 이라 가정합시다. 만일 $\hat{q} = m \oslash n$ 이라 정의한다면, 위 정리를 증명하기 위해서는

$$|n\hat{q} - m| \leq \frac{1}{4} \tag{1.9}$$

임을 보이면 충분합니다. 왜냐하면 m 은 소수점 뒤로 최대 1 개의 비트만 있으므로 $n\hat{q}$ 는 자동으로 m 으로 반올림 될 것이기 때문입니다. 예외적으로 $|n\hat{q} - m| = 1/4$ 가 되는 경우를 살펴보자면, 앞서 조건에서 조정하기전 m 은 $|m| < 2^{p-1}$ 이였기 때문에, 조정된 이후의 m 의 마지막 비트 역시 0 이 됩니다. 따라서 절반으로 나뉜 경우 역시 m 으로 반올림 됩니다. (항상 짝수로 반올림 됨)

이제 $q = .q_1q_2...$, 이라 정의하고, $\hat{q} = .q_1q_2...q_p1$ 이라 정의합시다. 이제 $|n\hat{q} - m|$ 을 계산하기 위해서는 먼저

$$|\hat{q} - q| = |N/2^{p+1} - m/n|$$

을 계산하도록 합시다.. 이 때, N 은 홀수인 정수 이다. $n = 2^i + 2^j$ 이고 $2^{p-1} \leq n < 2^p$ 이므로 어떤 $k \leq p-2$ 에 대해 반드시 $n = 2^{p-1} + 2^k$ 가 되어야만 합니다. 따라서

$$|\hat{q} - q| = |\frac{nN - 2^{p+1}m}{n2^{p+1}}| = |\frac{(2^{p-1-k} + 1)N - 2^{p+1-k}m}{n2^{p+1-k}}|$$

여기서 분자는 당연히 정수이고, N 이 홀수 이기 때문에, 분자는 홀수가 됩니다. 따라서,

$$|\hat{q} - q| \leq \frac{1}{n2^{p+1-k}}$$

가 성립합니다. 이 때 $q < \hat{q}$ 라 가정하면 (그 반대도 역시 비슷하다), $n\hat{q} < m$ 이 성립하고, 따라서

$$\begin{aligned} |m - n\hat{q}| = m - n\hat{q} = n(q - \hat{q}) &\leq n(q - (\hat{q} - 2^{-p-1})) \\ &\leq n(2^{-p-1} - \tfrac{1}{n2^{p+1-k}}) \\ &= (2^{p-1} + 2^k)2^{-p-1} - 2^{-p-1+k} = \tfrac{1}{4} \end{aligned}$$

따라서 정리가 증명됩니다. □

위 정리는 2 가 아닌 임의의 밑수 β 에 대해서도 $2^i + 2^j$ 가 아닌 $\beta^i + \beta^j$ 에 대해서 성립하게 됩니다. 다만, β 가 커지게 되면, 그 꼴의 좀더 드물어 질 뿐입니다.

우리는 이제 다음 질문에 답할 수 있게 되었습니다. "과연, 기초 연산에서 필요한 것 보다 살짝 조금 더 반올림 오차가 발생하는 것에 대해 신경을 써야 할까?" 그에 대한 답은, 신경을 써야 한다 이고, 왜냐하면 기초 연산을 정확하게 하는 일이야 말로 여러 공

식들이 작은 상대 오차를 야기하도록 사용할 수 있기 때문입니다. 지워짐(Cancellation)에서 우리는 정확한 결과를 내기 위해 보호 숫자를 필요로 하는 여러가지 알고리즘에 대해 논의하였습니다. 만일 그 공식들에게 주어지는 입력값 자체에 불확실 성이 있다면 정리 1.4.1 과 1.4.2 의 내용들은 아마 쓸모가 없어질 것입니다. 착한 지워짐이었던 $x - y$ 가, x, y 자체가 근사값이 되다면 나쁜 지워짐이 될 수 있기 때문이죠. 하지만, 정확한 연산 자체는 부정확한 입력값에 대해서도 매우 유용한데, 왜냐하면 정리 1.5.2 과 1.5.3 에서 나타났듯이 정확한 관계를 구축할 수 있기 때문입니다. 이들은 비록 부동소수점 변수가 참값의 근사치라도 매우 유용하게 사용할 수 있습니다.

제 2 장

IEEE 표준

IEEE 에는 두 개의 부동 소수점 연산 표준이 있습니다. IEEE 754 는 이진 표준으로 단정밀도(single precision) 에는 $\beta = 2, p = 24$를, 배정밀도(double precision) 에는 $p = 53$ 을 요구합니다 [IEEE 1987]. 또한 이 표준에는 단정밀도와 배정밀도에서 어떠한 식으로 비트를 배열할지도 명시하고 있습니다. IEEE 854 는 754 와는 달리 $\beta = 2$ 와 $\beta = 10$ 을 허용하고, 어떠한 식으로 부동 소수점 수가 비트들로 인코딩 되야 하는지 명시하지 않았습니다 [Cody et al. 1984]. 또한 명확히 p 로 어떠한 값을 써야하는지 지정하지는 않았지만, 단, 배정밀도에서 가능한 p 값들을 제한해놓았습니다. 앞으로 **IEEE 표준** 이란 말은 두 개의 표준을 모두 지칭하는데 사용하도록 하겠습니다.

이 장에서는 IEEE 표준에 대해 간단하게 짚고 넘어가도록 하겠습니다. 각각의 섹션에서는 IEEE 표준의 한 측면에 대해 살펴보도록 하며, 왜 이러한 것을 정했는지도 이야기 하겠습니다. 이 문서는 IEEE 표준이 가장 좋은 부동 소수점 표준이라고 주장하기 위해 만들어 진 것이 아니라, 단지 어떻게 하면 IEEE 표준을 잘 이해하고 사용할 수 있는 지 설명해주는 것입니다. 더 자세하게 알고 싶은 분들은 IEEE 표준 문서를 직접 참고하시기 바랍니다. [IEEE 1987; Cody et al. 1984]

2.1 형식과 연산들

2.1.1 밑수(base)

왜 IEEE 854 에서 $\beta = 10$ 을 허용하는지는 명백합니다. 우리 인간이 십진수를 사용해서 계산을 하기 때문입니다. 따라서 $\beta = 10$ 은 특히 계산기에서 사용되기에 매우 적합한데, 왜냐하면 계산기에서 각각의 계산 결과는 모두 십진수로 표시되기 때문입니다.

하지만 IEEE 854 에서 밑이 10 이 아니라면 반드시 2 를 사용하도록 정해 놓은 이유는 있습니다. 상대 오차와 Ulp 에서 한 가지 이유를 이야기 했었었는데. 왜냐하면 .5ulps 의 반올림 오차에 해당하는 상대오차가 β 의 배수로 증가되기 때문입니다. 뿐만 아니라, 밑수가 커지게 된다면 상대적으로 유효 정밀도가 떨어진다는 점이 있는데, 예를 들어 $\beta = 16, p = 1$ 인 부동 소수점과 $\beta = 2, p = 4$ 인 경우를 각각 살펴보도록

합시다. 이 둘은 모두 동일하게 4 비트의 가수를 가지게 됩니다. 이를 이용해서 15/8 을 계산한다고 생각해봅시다. $\beta = 2$ 인 경우에 15 는 1.111×2^3 으로 표현되고, 15/8 은 1.111×2^0 이 되서 정확하게 계산됩니다. 하지만 $\beta = 16$ 일 때 15 는 $F \times 16^0$ 이고 (F 는 16 진수로 15) 15/8 은 1×16^0 로 오직 1 개 비트만 정확하게 나옵니다. 일반적으로 16 진수는 최대 3 개의 비트를 손실하게 되므로, p 16 진수 자리수는 유효 정밀도가 이진수의 경우 $4p$ 였겠지만 $4p-3$ 비트까지 떨어지게 됩니다. 따라서 β 가 커지게 되면 이러한 문제들이 발생하게 되는데요, 그럼 왜 IBM 은 그들의 system/370 시스템에 $\beta = 16$ 을 사용하였을까요. 사실 정확한 이유는 IBM 만이 알고 있겠지만, 아마 2 개의 이유를 들 수 있겠습니다. 첫 번째 이유는 더 넓은 범위의 지수를 이용할 수 있는 점인데요, IBM system/370 의 단정밀도는 $\beta = 16, p = 6$ 을 사용하였습니다. 따라서 가수 부분은 24 비트이고, 7 비트를 지수 부분이 차지하게 됐는데, 이를 통해서 16^{-2^6} 에서 $16^{2^6} = 2^{2^8}$ 까지의 넓은 범위의 수를 사용할 수 있게 되었습니다. $\beta = 2$ 일 때 동일한 지수 범위를 사용하려면 9 비트를 지수로 사용해야 하고, 가수로 오직 22 비트 밖에 사용할 수 없게 좁니다. 하지만, 앞에서도 말했지만 $\beta = 16$ 일 때 유효 정밀도가 최대 21 비트 까지 떨어지므로 오히려 $\beta = 2$ 를 사용했을 안정적으로 23 비트의 정밀도 (22 비트의 가수지만 뒤에서 설명하겠지만 추가적인 비트로 정밀도를 한 비트 더 얻을 수 있음) 를 얻을 수 있게 되어서 더 안정적입니다.

$\beta = 16$ 을 고른 이유로 또 가능한 것은 바로 쉬프트 연산 때문입니다. 두 개의 부동 소수점 수를 더할 때, 만일 그들의 지수들이 다르다면, 둘 중 하나의 가수들이 쉬프트 되어서 소수점을 맞춰야 하므로 연산 속도가 느려지게 됩니다. $\beta = 16, p = 1$ 시스템 에서는 1 부터 15 까지의 숫자들이 모두 같은 지수를 가지게 되므로, $\binom{15}{2} = 105$ 가지의 쌍에 대해 덧셈 시에 쉬프트 연산을 할 필요가 없게 됩니다. 반면에 $\beta = 2, p = 4$ 시스템에서는 1 부터 15 까지의 숫자들의 지수들이 0 부터 3 까지에 걸쳐 있으므로 105 개의 쌍 중에서 70 개의 쌍 사이에서 쉬프트 연산이 필요하게 됩니다.

하지만 대부분의 현대 하드웨어에서는 쉬프트 연산으로 이한 작업 저하 정도가 매우 미미하기 때문에 워블을 작게 줄여주는 $\beta = 2$ 를 이용하는 경우가 많습니다. 특히 $\beta = 2$ 를 이용하게 된다면 1 개의 추가적인 비트를 가수에 더 사용할 수 있기 때문에[1] 정밀도를 더 높일 수 있습니다. 모든 부동 소수점 수가 언제나 정규화 되기 때문에, 가수의 가장 최고 비트 (맨 왼쪽) 는 언제나 1 이 되므로 굳이 이 비트를 저장하기 위해 비트를 낭비할 필요가 없게 됩니다. 이러한 트릭을 이용하는 것을, 숨겨진 비트(hidden bit) 를 사용한다고 말합니다. 하지만 이 트릭을 이용하기 위해서는 부동소수점 형식 (Floating-point Formats)에서 지적하였듯이, 0 을 위한 특별한 처리가 필요하게 됩니다. 따라서 0 을 표현하기 위해 지수가 $e_{min} - 1$ 이고, 가수가 모두 0 이라면, $1.0 \times 2^{e_{min}-1}$ 이 아니라, 0 을 나타내도록 정하였습니다.

IEEE 754 단정밀도는 32 비트로 구성되며, 1 비트는 부호비트, 8 비트는 지수 비트, 그리고 나머지 23 비트는 가수 비트를 위해 사용됩니다. 하지만 숨겨진 비트를 이용하여,

1) 이 사실은 비록 Knuth ([1981], 211 쪽) 가 Konard Zuse 의 공으로 돌렸지만 사실 Goldberd [1967] 이 최초로 발표하였습니다.

인자	형식			
	단정밀도	확장 단정밀도	배정밀도	확장 배정밀도
p	24	≥ 32	53	≥ 64
e_{max}	+127	≥ 1023	+1023	> 16383
e_{min}	-126	≤ -1022	-1022	≤ -16382
지수 비트 수	8	≤ 11	11	≥ 15
형식 전체 비트 수	32	≥ 43	64	≥ 79

표 2.1: 표 2.1.2 IEEE 형식 인자들

비록 23 비트로 인코드 되지만, 실질적인 가수 부분은 24 비트 ($p = 24$) 가 됩니다.

2.1.2 정밀도 (Precision)

IEEE 표준에서는 4 개의 서로 다른 정밀도를 지정합니다. 단정밀도, 배정밀도, 확장 단정밀도 (single-extended), 확장 배정밀도 (double-extended). IEEE 754 에서는 대부분의 부동 소수점 하드웨어에서 제공하는 단, 배정밀도와 대응됩니다. 단정밀도는 32 비트 워드를 사용하고, 배정밀도는 두 개의 인접한 32 비트 워드를 사용하게 됩니다. 확장 정밀도 형식은 약간 더 늘어난 정밀도와 지수 범위를 지원합니다. (표 2.1.2 참고)

확장 정밀도의 경우, IEEE 표준은 얼마나 많은 비트들을 더 추가해야 하는지에 대한 하한선 만을 지정하였습니다. 확장 배정밀도의 경우 최소한 80 비트 이상을 사용하도록 정하였는데, 표 2.1.2 에 나온 것은 79 비트 만으로 구현해도 충분합니다. 그럼 1 비트를 더 추가적으로 필요로 하는 이유는, 대부분의 확장 정밀도를 지원하는 하드웨어들이 숨겨진 비트를 사용하지 않기 때문에 79 비트 만으로 충분하지만 최소 80 비트 이상을 요구하고 있습니다.

표준안은 확장 정밀도에 좀 더 많은 강조를 하였는데 (반면에 배정밀도 지원은 특별히 강조하지 않았지만), "표준안 구현시에 확장 형식을 꼭 지원하도록 강력히 추천한다*(Implementations should support the extended format corresponding to the widest basic format supported..)*" 라며 사용을 권장하였습니다.

이렇게 확장 정밀도 사용을 도입한 한 가지 중요한 계기는 바로 계산기 때문입니다. 계산기들은 보통 10 개의 숫자들만 표시하지만, 내부적으로 13 개의 숫자를 사용해서 처리하기 때문입니다. 계산기의 경우, exp 나 log 와 같은 함수들을 10 자리 정확도로 계산하기 위해서는, 몇 개의 자리수가 더 필요하게 됩니다. 따라서 내부적으로 13 개의 자리수를 이용해서 계산하되, 보여줄 때는 정확한 10 개 비트만 보여주게 됩니다.

IEEE 표준의 확장 정밀도 역시 이와 비슷한 역할을 합니다. 이를 이용해서 라이브러리로 하여금 .5ulp 오차 이내로 단정밀도 (혹은 배정밀도) 계산을 수행할 수 있게 됩니다. 하지만 확장 정밀도 사용시 중요한 것이 있는데, 바로 이를 사용시에 반드시 사용자에게 투명하게 공개해야 된다는 것입니다. 예를 들어 계산기의 경우 내부에서 처리된 값이 화면에 보여지는 값과 동일한 정밀도로 반올림 되지 않는다면, 그 값을 이용해서 계산을 계속 하였을 때, 내부에 숨겨진 숫자들 때문에 유저들의 입장에서는 불일치 하게 나타날 수 있기 때문입니다.

확장 정밀도를 더 설명하기 위해, IEEE 754 단정밀도와 10진수 사이의 변환 문제에 대해 생각해보도록 합시다. 이상적으로, 단정밀도 수들은 표시될 때 충분한 십진수 숫자로 표시되므로, 다시 십진수로 읽어들일 때 원래의 단정밀도로 복구할 수 있게 됩니다. 사실 9 자리의 십진수만 이용해도 유일한 단정밀도 이진수로 복구할 수 있다는 것이 알려져 있습니다 (이진수를 십진수로 변환하기 참조) 어떤 십진수를 다시 유일한 이진 표현으로 변경할 때, 1ulp 정도로 작은 반올림 오차 조차 치명적일 수 있습니다. 왜냐하면 이는 아예 다른 답이 되기 때문이지요. 따라서 이 때 효율적인 알고리즘을 위해서 확장 정밀도를 사용하는 것입니다. 만일 확장 단정밀도가 사용 가능하다면 십진수를 이진수로 바꾸는 매우 직관적인 방법이 있습니다. 먼저 9 개의 십진숫자들을 N 이라고 합시다. (소수점을 무시하고 읽는다) 그럼 표 2.1.2 에서 $p \geq 32$ 이고, $10^9 < 2^{32} \approx 4.3 \times 10^9$ 이므로 N 은 정확히 확장 단정밀도로 표현될 수 있습니다. 이제, 원래 숫자의 소수점 위치를 맞추기 위해 적절한 10^p 을 찾습니다. 이제 $10^{|p|}$ 을 계산하는데, 만일 $|P| \leq 13$ 이라면, 이 역시 정확하게 변환될 수 있습니다. 왜냐하면 $10^{13} = 2^{13}5^{13}$ 이고 $5^{13} < 2^{32}$ 이기 때문이죠. 이제 마지막으로, N 과 $10^{|P|}$ 를 곱하면 됩니다. 만일 마지막 작업이 정확히 수행되었다면, 가장 가까운 이진수로 복구할 수 있게 됩니다. 이진수를 십진수로 변환하기을 보면 어떻게 마지막 곱셈을 수행하는지 설명되어 있습니다. 따라서 $|P| \leq 13$ 일 때, 확장 단정밀도 형식을 사용하는 것은 9 자리 십진수를 정확히 가장 가까운 이진수로 변환하는 것이 가능합니다 (단, 정확히 반올림 된 경우에). 만일 $|P| > 13$ 이라면 확장 단정밀도로는 항상 정확히 반올림 된 이진수로 변환하는 것이 위 알고리즘으로는 충분하지 않습니다. 하지만 Coonen [1984] 은 이진수를 십진수로 변환하고 다시 원래 이진수로 복구하는데에는 충분하다는 사실을 보였습니다.

만일 배정밀도가 지원 된다면, 위 알고리즘은 확장 단정밀도 보다는 배정밀도에서 수행하여도 됩니다. 하지만 배정밀도를 17 자리 십진수로 변환하고 다시 역변환 하는 작업은 확장 배정밀도를 필요로 합니다.

2.1.3 지수

지수 자체가 양수 혹은 음수 둘 다 될 수 있기 때문에 부호를 나타내기 위해 특별한 방법을 필요로 합니다. 보통 부호 있는 수를 나타내는 방법으로 두 가지 방법이 있는데 하나는 부호와 그 절대값을 따로 보관 (sign/magnitude) 하는 방법과 2 의 보수 표현 (2's complement) 방식이 있습니다. 부호 비트를 따로 보관하는 방법은 IEEE 형식에서 가수 부분을 처리할 대 사용하는 것입니다. 즉 1 개 비트를 부호를 나타내는데 사용하고, 나머지 비트를 그 수의 크기 (절대값) 을 나타내는데 사용하는 것입니다. 한편 2 의 보수 표현은 보통 정수 연산에서 자주 쓰이게 됩니다. 이 표현 방식을 사용하면 $[-2^{p-1}, 2^{p-1} - 1]$ 범위의 수들을 모듈로 2^p 로 나오는 가장 작은 음이 아는 정수로 표현하게 됩니다.

bias 의 사전적 의미는 편향, 치우침으로, 결과적으로 지수값에 127 을 뺌으로써 그 값이 **치우쳐져 있다** 라는 맥락에서 이름 붙인 것이라 생각하시면 됩니다.

IEEE 이진 표준은 지수를 처리하기 위해 이 두 방식 모두 사용하지 않고, 그 대신에 바이어스(biased) 표현을 사용합니다. 예를 들어 단정밀도에서, 지수 부분은 8 비트로

저장되게 되는데, 이 때 바이어스는 127 입니다 (배정밀도는 1023). 이 말은, 만일 $\bar{k}$ 가 부호없는 정수로 표현된 지수 비트라면, 실제로 부동 소수점 수의 지수 값은 $\bar{k} - 127$ 이라는 뜻입니다. 이 실제 지수 값은 바이어스 된 지수인 $\bar{k}$ 와 구분하기 위해 바이어스 되지 않은 지수(unbiased exponent) 라고 부릅니다.

표 2.1.2 에 딷르면 단정밀도는 $e_{max} = 127$ 이고 $e_{min} = -126$ 이었습니다. 왜 $|e_{min}| < e_{max}$ 가 되도록 처리하였다면, 가장 작은 수의 역수가 $(\frac{1}{2^{e_{min}}})$ 오버플로우(overflow) 나지 않게 하기 위함이었습니다. 비록 가장 큰 수의 역수가 언더플로우(underflow) 날 수 있지만, 많은 경우 언더플로우가 오버플로우 보다는 덜 심각한 문제이기 때문입니다. 밑수(base) 에서 $e_{min} - 1$ 이 0 을 표현하기 위해 사용된다고 하였고, 특별한 수들 섹션에서 $e_{max} + 1$ 의 사용 예들을 볼 것이빈다. IEEE 단정밀도에서는, 바이어스 된 지수들의 범위가 $e_{min} - 1 = -127$ 에서 $e_{max} + 1 = 128$ 이라는 것이고, 다시 말해 바이어스 되지 않는 지수들의 경우 그 범위가 0 에서 255 까지라는 의미입니다. 이는 정확히 부호 없는 8비트 이진수가 표현할 수 있는 수들이지요.

오버플로우는(overflow)는 단어 그대로, 수의 크기가 너무 커서 보관할 수 있는 범위를 초과해 버린다는 의미 입니다. 예를 들어서 4비트 정수형을 보관하는 자료형이 있다면, 최대 크기는 1111 인데, 만일 여기에 1 을 더하게 된다면, 오버플로우가 발생하며, 0000 이 되버립니다. 이는 엄청난 계산상의 문제가 되겠지요. 언더플로우(underflow) 는 이 정 반대의 경우 입니다.

2.1.4 연산들(operations)

IEEE 표준에서는 덧셈, 뺄셈, 곱셈, 그리고 나눗셈의 결과가 정확히 반올림 되기를 요구하고 있습니다. 즉, 연산이 먼저 정확히 계산이 된 다음, 가장 가까운 부동 소수점 수로 반올림 (짝수 반올림 방식을 사용해서) 되어야 한다는 말입니다. 이전의 보호 숫자 (Guard Digits)에서, 두 개의 부동 소수점들의 지수 차이가 매우 클 때 두 수의 합 이나 차이를 계산하는 일은 매우 어려운 일이라는 것을 말했었습니다. 따라서 보호 숫자를 도입해서 작은 상대오차를 유지하면서 효과적으로 계산하였다고 하였습니다. 하지만, 한 개의 보호 숫자를 이용해서 계산하는 것은 정확히 계산한 다음에 반올림 하는 것과 언제나 같은 결과를 낼 것이라 보장할 수 없습니다. 하지만 두번째 보호 숫자와, 세번째 스티키 비트(sticky bit) 를 도입해 총 3 개의 보호 숫자를 이용해 한 개의 보호숫자를 이용해서 계산하는 것 보다 살짝 더 많은 비용으로 정확히 계산한 후 반올림 하는 효과를 낼 수 있습니다. [Goldberg 1990] 따라서 이를 통해 표준안이 효율적으로 구현될 수 있지요.

산술 연산 결과를 명확히 지정하는 이유는 바로 프로그램들의 이식성(portability)을 위해서 입니다. 같은 IEEE 산술 표준을 사용하는 두 개의 기계 사이에서 작동하는 프로그램은 반드시 같은 결과를 내도록 표준안이 정해져 있으며, 만일 그 결과가 다르다면 산술 연산에서의 차이가 아니라 소프트웨어의 버그라고 할 수 있게 됩니다. 세세하게 표준안을 정해놓은 또 다른 이유로, IEEE 산술에서 산술적으로 올바른 값을 낸 다고 증명된 것들을 IEEE 표준을 지원하는 기계에서 정확히 작동할 것이라 보장할 수 있기 때문입니다.

Brown [1981] 은 모든 부동 소수점 하드웨어에 사용 가능한 공리(axiom)들을 제안하였습니다. 하지만 이 공리들은 지워짐(Cancellation) 이나 정확한 반올림 연산 에서 다루어 왔던 유용한 알고리즘들을 증명 할 수 없었는데, 왜냐하면 이들은 하드웨어에 공통적으로 들어 있지 않는 특별한 기능들(보호 숫자 연산 등)을 필요로 하기 때문이

었습니다. 게다가 Brown 의 공리들은 연산이 정확히 수행되고 반올림 되는지 단순히 정의하는 것 보다 훨씬 복잡하였습니다. 따라서 Brown 의 공리들로 부터 정리들을 증명하는 일은, 단순히 작업들이 정확히 반올림 된다고 가정하고 증명하는 일 보다 훨씬 어렵습니다.

부동 소수점 표준에서 어느 연산까지 다루어야 할 지에 대해서는 아직 의견이 일치가 되지 않았습니다. 게다가, IEEE 표준에서는 기본적인 사칙 연산 외에도, 제곱근, 나머지 연산(modular), 그리고 정확히 반올림 되는 정수와 부동 소수점 수와의 변환 등에 대해 명시하고 있습니다. 뿐만 아니라 표준에서는 내부 형식과 십진수 사이의 변환이 정확히 반올림 되도록 요구하였습니다. (아주 큰 수들은 제외) Kulish 와 Mirnaker [1986] 은 표준안에 내적 연산을 포함시켜서 그 연산 과정을 정확히 명시해달라고 제안하였습니다. 그들은 IEEE 산술 연산으로 이용해서 내적 연사을 계산했을 때, 최종 답안이 꽤 틀릴 수 있기 때문이라고 주장하였습니다.

IEEE 표준안은 표 제작자의 딜레마 (table maker's dilemma) 때문에 초월 함수들의 값들이 정확히 반올림 되도록 요구하지 않습니다. 예를 들어 여러분이 지수 함수의 표를 4 자리 정밀도로 만든다고 합시다. 그렇다면 $\exp(1.626) = 5.0835$ 일 때 이를 반올림을 5.083 으로 해야할 까요, 아니면 5.084 로 해야 할까요. 사실 $\exp(1.626)$ 을 좀 더 적확히 계산한다면, 5.08350 이 됩니다. 더 정확히 하면 5.084500. 더 정확히 하면 5.0845000. 사실 exp 함수가 초월함수이기 대문에 $exp(1.626)$ 값을 정확히 반올림 하기 위해서는 5.08350000000...ddd 가 될지 5.0834999...9ddd 가 될 지 모르기 때문에 무한히 많은 연산이 필요하질도 모릅니다. 따라서, IEEE는 초월함수들의 정밀도를 정확하게 구하는 것은 무의미 하다고 판단하였습니다. 이 문제를 해결하기 위해 제안된 방법으로 표준안에서 이러한 초월 함수들의 값을 구하는 알고리즘들을 명확히 지정하는 방법이 있는데, 사실 모든 하드웨어 아키텍쳐 위에서 동일하게 잘 작동하는 단일 알고리즘은 없기 대문에 결과적으로는 표준안에서 초월함수에 대한 어떠한 필수적인 요건은 갖추지 않게 되었습니다. 유리 근사(Rational approximation), CORDIC[2], 그리고 거대한 표를 이용하는 것이 현대에 컴퓨터들에서 초월함수 값을 계산하는 기본적인 방법이 되었습니다. 각각의 방법들은 하드웨어에 의존적이며, 모든 하드웨어에 대해 잘 작동하는 알고리즘은 아직까지 없습니다.

2.2 특별한 값들

몇몇의 부동 소수점 하드웨어에서는 모든 비트 패턴들이 하나의 유효한 부동 소수점 수를 나타냅니다. 예를 들면 IBM 의 System/370 을 들을 수 있죠. 하지만 VAM^{tm} 의 경우, 일부 비트 패턴들을 특별한 수를 나타내기 위해서 예약해 놓았는데, 이들을 예약된 피연산자(reverved operand) 라고 부릅니다. 이 아이디어는 CDC 6600 으로 거슬러 올라가는데, 이 시스템에서 미정와 무한대를 나타내기 위해 특별한 값들 -

2) CORDIC 은 Coordinate Rotation Digial Comptuer 의 약자로, 쉬프트와 덧셈 연산을 많이 사용하는 (그리고 극 소수의 덧셈과 나눗셈) 초월 함수 계산 방법입니다 [Walther 1971].

지수	가수	나타내는 값
$e = e_{min} - 1$	$f = 0$	± 0
$e = e_{min} - 1$	$f \neq 0$	$0.f \times 2^{e_{min}}$
$e_{min} \leq e \leq e_{max}$	-	$1.f \times 2^{e}$
$e = e_{max} + 1$	$f = 0$	$\pm\infty$
$e = e_{max} + 1$	$f \neq 0$	NaN

표 2.2: 표 2.2 IEEE 754 의 특별한 값들

INDEFINITE, INFINITY 를 사용하였습니다.

IEEE 표준에서는 이 전통을 계승하여서 NaN (Not a Number - 수가 아님) 과 무한대를 도입하였습니다. 이러한 특별한 값들을 도입하지 않는한, 음수의 제곱근을 계산하는 것과 같은 예외 상황을 연산을 중지시키지 않는 한 잘 처리할 수 없게 됩니다. IBM System/370 FORTRAN 의 경우, 음수의 제곱근과 같은 명령을 처리할 때 기본값으로 수행하는 명령은 오류 메세지를 표현하는 것입니다. 게다가 이 시스템에서는 모든 비트 패턴이 하나의 유효한 수이기 때문에, 예를 들어 -4 제곱근 역시 어떤 부동 소수점 값을 리턴하게 됩니다. 이 System/370 FORTRAN 의 경우 $\sqrt{-4} = 2$ 를 리턴하였습니다. IEEE 산술에서는 이러한 상황에서 NaN 이 리턴됩니다.

IEEE 표준안에서는 다음과 같은 몇 개의 특별한 값들을 정해놓았습니다. (표 2.2 참조) ± 0, 비 정규화 된 수, $\pm\infty$, 그리고 NaN 들 입니다 (다음장에서 설명하겠지만 NaN 은 한 개가 아닙니다) 이러한 특별한 값들의 지수부분은 $e_{max} + 1$ 이거나 $e_{min} - 1$ 을 가지게 됩니다.

2.2.1 NaN 들

전통적으로 $0/0$ 이나 $\sqrt{-1}$ 과 같은 계산들은 복구할 수 없는 오류로, 계산을 중단시키게 만들었습니다. 하지만, 이러한 값들이 나타나도 계산을 계속해야만 하는 몇 가지 상황들이 있는데요, 예를 들어서 함수의 해(zero) 를 찾는 루틴을 만들었다고 합시다. 어떤 함수 f 의 해를 찾는 $zero(f)$ 라는 함수가 있다면, 전통적으로 이 함수는 f 가 정의되어 있는 구간 $[a, b]$ 안에서 해를 찾도록 구간에 대한 정보를 따로 요구를 할 것입니다. 따라서, 해 찾는 함수를 정확히 말하자면 $zero(f, a, b)$ 라고 할 수 있겠습니다. 하지만 유용한 해 찾는 함수라면, 구간 $[a, b]$ 에 대한 정보와 같은 추가적인 정보를 필요로 하지는 않겠지요. 우리가 해를 찾는 함수를 만들었을 때 그 함수가 정의된 구간까지 따로 입력해 준다면 매우 귀찮은 일이 아닐 수 없습니다. 하지만, 이렇게 $zero$ 힘수가 해를 찾기위해 탐색을 할 때, 만일 f 의 정의역을 벗어나는 값을 넣게 된다면, 아마도 함수값을 계산하는 도중에 $0/0$ 이나 $\sqrt{-1}$ 과 같은 것들을 계산할 것이며, 이 때 전체 계산과정을 중단시키고 불필요하게도 해 찾는 과정 역시 중단되게 되겠지요. 이렇나 문제는 NaN 이라 불리는 특별한 값을 도입함으로서 해결할 수 있습니다. $0/0$ 이나 $\sqrt{-1}$ 과 같은 것을 연산하였을 때 계산을 중단시키기 보다는 NaN 을 리턴하도록 말이지요. NaN 이 발생할 수 있는 사례들은 에 잘 나타나 있습니다. 따라서 $zero(f)$ 가 f 의 정의역 밖을 탐색하더라도, 이 함수는 NaN 을 리턴하여서, 함수가 계산을

연산	NaN 이 만들어지는 경우
$+$	$\infty + (-\infty)$
$\times$	$0 \times \infty$
$/$	$0/0, \infty/\infty$
REM	x REM 0, ∞ REM y
$\sqrt{\bullet}$	$\sqrt{x}, x < 0$

표 2.3: 표 2.2.1 NaN 을 발생시키는 연산들

중지시키지 않고 계속 찾아 나갈 수 있게 된다는 것이지요. 이러한 사실을 염두에 둘 때, 우리는 NaN 과 보통의 정상적인 부동 소수점 수 와의 연산을 했을 때 어떠한 결과가 나올 지 생각할 수 있습니다. 예를 들어 f 가 `return(-b + sqrt(d))/(2*a)` 와 같이 정해졌다고 해봅시다. 만일 $d < 0$ 이라면 f 는 NaN 을 리턴하게 됩니다. 왜냐하면 $d < 0$ 이면 `sqrt(d)` 가 NaN 이 되므로 $-b + sqrt(d)$ 가 NaN 되도록 해야 하기 때문입니다. 비슷하게도, 나눗셈시 만일 하나의 피연산자가 NaN 이라면 그 나눈 결과 역시 NaN 이 됩니다. 일반적으로 NaN 이 어떤 부동 소수점 연산을 하게 되면 그 결과는 NaN 이 됩니다.

사용자가 정의역 범위를 지정하지 않아도 안전하게 해를 찾는 함수를 설계하는 또 하나의 방법은 바로 시그널(signal) 을 사용하는 것입니다. 해 찾는 함수 안에 부동 소수점 예외를 처리하기 위한 시그널 핸들러(signal handler) 을 도입한 다음에, 만일 f 를 계산하는데 정의역을 벗어난 값을 계산해서 부동 소수점 예외가 발생하였다면, 시그널 처리기에서 이를 받아 처리한 후 다시 계산을 지속할 수 있습니다. 다만 이 방식을 도입하였을 때 문제점은, 모든 언어에서 각기 다른 방법으로 시그널을 처리하는 방법이 있기 때문에 (혹은 아예 없는 경우도 있다), 높은 이식성을 보여주지는 못합니다.

IEEE 754 에서 NaN 은 주로 지수가 $e_{max}+1$ 이고 0 이 아닌 가수를 가지는 값으로 구현됩니다. 정확히 가수에 어떠한 값이 들어가야 하는 문제는 시스템에 따라 다르게 설계되므로, 한 개의 유일한 NaN 이 존재하는 것은 아니고, NaN 계열들이 존재하게 됩니다. 다만 NaN 과 어떤 부동 소수점 수와 연산하였을 때 그 결과 역시 NaN 이 되어야만 합니다. 따라서 만일 어떠한 긴 연산의 결과가 NaN 이였을 때, 그 시스템 독립적인 정보를 포함하고 있는 가수를 분석하여서 언제 가장 첫번째 NaN 이 발생하였는지를 알 수 있게 됩니다. 사실 이 마지막 문장에는 약간의 어폐가 있는데, 만일 두 피연산자 모두 NaN 이라면 그 연산 결과는 둘 중 하나의 NaN 이므로 가장 먼저 만들어진 NaN 을 추적할 수 는 없습니다.

2.2.2 무한대

NaN 이 $0/0$ 이나 $\sqrt{-1}$ 과 같은 것들을 처리하는데 사용되었다면, 무한대는 오버플로우가 발생하였을 때 처리하는데 사용됩니다. 무한대를 따로 도입하는 것이 단순히 표현 가능한 최대의 수를 사용하는 것 보다 더 안전한 방법입니다. 예를 들어서 $\sqrt{x^2+y^2}$ 를 계산한다고 해봅시다. 만일 $\beta = 10, p = 3, e_{max} = 98$ 이라고 하고 $x = 3 \times 10^{70}$ 이고, $y = 4 \times 10^{70}$ 이라고 한다면 x^2 는 오버플로우가 발생해서 9.99×10^{98}

이 될 것입니다. y 역시 마찬가지로 $x^2 + y^2$ 계산시 각각은 모두 오버플로우가 나서 전체 결과 역시 오버플로우가 발생한 9.99×10^{98} 이 됩니다. 따라서 최종 연산 결과는 $\sqrt{9.99 \times 10^{98}} = 3.16 \times 10^{49}$ 이 되고 이는 참값인 5×10^{70} 과 전혀 다르게 나옵니다. 반면에 IEEE 산술을 하게 된다면 x^2 은 ∞ 이고, 마찬가지로 y^2 역시 ∞ 라서, 최종 결과는 ∞ 가 됩니다. 차라리 이렇게 처리하는 것이, 참값과 전혀 다른 부동 소수점 수를 리턴하는 것 보다 더 안전하다고 볼 수 있습니다.

0 을 0 으로 나누는 것은 NaN 을 발생시킵니다. 하지만 0 이 아닌 수를 0 으로 나누는 것은 무한대를 발생시킵니다. 즉, $1/0 = \infty, -1/0 = -\infty$ 가 됩니다. 이 두 개의 상황을 구분짓는 이유는 다음과 같습니다. 만일 $x \to 0$ 일 때, $f(x) \to 0, g(x) \to 0$ 이라고 합시다. 그러면 $f(x)/g(x)$ 는 어떠한 값도 가질 수 있게 됩니다. 예를 들어서 $f(x) = \sin(x), g(x) = x$ 라고 하면, $x \to 0$ 일 때 $f(x)/g(x) \to 1$ 이 됩니다. 반면에 $f(x) = 1 - \cos(x)$ 라면, $f(x)/g(x) \to 0$ 이 되죠. 따라서 0/0 을 생각해볼 때, 이는 그 어떤 값도 될 수 있습니다. 따라서 IEEE 표준에서는 이를 아예 NaN 이라고 처리하였습니다. 반면에 $c > 0$ 이고, 임의의 해석 함수 f, g 에 대해 $f(x) \to c, g(x) \to 0$ 이라면 $f(x)/g(x) \to \pm\infty$ 가 됩니다. 따라서 IEEE 표준에서는 $c \neq 0$ 인 이상 $c/0 = \pm\infty$ 가 되도록 정의하였습니다. 이 때 ∞ 의 부호는 c 와 0 의 부호에 따라가도록 하였습니다. 또한 사용자들은 플래그(flag)의 상태를 점검함으로써 ∞ 의 발생이 오버플로우로 인한 발생인지, 0 으로 나눔으로 인한 발생인지 확인할 수 있습니다. 전자의 경우에서는 오버플로우 플래그(overflow flag)가 설정이 되고, 후자의 경우에서는 제로 플래그(zero flag) 가 설정이 됩니다.

무한대를 가지고 있는 연산에서 그 결과를 결정하는 것은 꽤 같단합니다. 무한대에 해당하는 부분을 x 로 바꾸고, $x \to \infty$ 를 계산하면 되지요. 따라서 $3/\infty = 0$ 입니다. 왜냐하면

$$\lim_{x \to \infty} 3/x = 0$$

비슷하게도, $4 - \infty = -\infty$ 이고, $\sqrt{\infty} = \infty$ 가 되빈다. 하지만 극한값이 존재하지 않는 경우에 그 결과는 NaN 이 되는데, 따라서 ∞/∞ 는 NaN 이 됩니다. (표 2.2.1 에 여러 예들이 있습니다) 이러한 논리는 0/0 이 왜 NaN 이 되었는지와 일맥 상통합니다.

어떠한 것을 연산을 할 때 그 중간에 NaN 이 포함되어 있다면 전체 연산 결과 또한 NaN 이 됩니다. 하지만 $\pm\infty$ 의 경우, $1/\infty = 0$ 과 같은 규칙들 때문에 ∞ 가 포함되어 있다고 해도 그 연산결과가 부동 소수점 수가 될 수 있습니다. 예를 들어 $x/(x^2 + 1)$ 과 같은 함수를 생각해봅시다. 사실 이는 안좋은 식인데, 분모의 x 가 $\sqrt{\beta}\beta^{e_{max}/2}$ 보다 커지게 된다면 오버플로우가 발생해서 $1/x$ 에 가까운 값이 아니라 0 이라는 값을 출력하게 됩니다. 하지만 위 식을 다시 쓰면 $1/(x + x^{-1})$ 로 쓸 수 있는데, 이를 통해 너무 일찍 오버플로우가 발생하는 일을 막을 수 있게 됩니다. 또한 $x = 0$ 이여도 $x = 0 : 1/(0 + 0^{-1}) = 1/(0 + \infty) = 1/\infty = 0$ 이라는 올바른 결과를 내게 됩니다. 만일 무한대 연산이 없었다면 이와 같이 식을 변형하였을 대 따로 $x = 0$ 인지를 확인을 해야 겠었지요. 무한대 연산이 있는 덕택에 여러 특별한 경우에 따로 체크를 해 줄 필요가 없어졌습니다. 하지만 그 대신에 $x/(x^2+1)$ 처럼 무한대에서 이상한 결과를 내지 않는지

확인해 주어야만 합니다.

2.2.3 부호 있는 영 (signed zero)

0 은 지수에 $e_{min}-1$ 을 넣고 가수 부분이 0 인 것으로 표현됩니다. 부호 비트에 2 개의 서로다른 값이 들어갈 수 있기 때문에 $+0, -0$ 두 가지의 0 이 가능합니다. 만일 $+0$ 과 -0 가 서로 다른 것으로 취급된다면, 매우 간단한 `if(x == 0)` 과 같은 작업들조차 x 의 부호에 따라 매우 불확실한 결과를 내게 됩니다. 따라서 IEEE 표준에서는 $-0 < +0$ 이 아니라 $+0 = -0$ 이라고 정해 놓았습니다. 물론 0 의 부호 자체를 무시하는 방법도 있지만 IEEE 표준에서는 그렇게 하지 않았습니다. 왜냐하면, 부호 있는 0 을 포함한 곱셈이나 나눗셈 시행시에, 보통의 수에서 부호를 처리하듯이 계산 결과가 나타나기 때문이지요. 따라서 $3/(+0) = +0, +0/-3 = -0$ 이 됩니다. 만일 0 에 부호가 없었다면 $1/(1/x) = x$ 와 같은 관계는 $x = \pm\infty$ 일 때 성립하지 않게 됩니다. 왜냐하면 0 에 부호를 두지 않는다면 $1/-\infty, 1/+\infty$ 모두 0 이 되는데, $1/0$ 은 $+\infty$ 이기 때문이지요. 따라서 0 에 부호가 없다면 $1/(1/x) = x$ 와 같은 식을 계산하는 과정에서 오버플로우 된 정보의 부호를 읽어버리게 되는 결과를 야기할 수 있습니다. 부호 있는 0 을 사용하는 다른 이유로 언더플로우와 0 에서 로그함수와 같은 불연속적인 함수를 다룰 때 사용됩니다. IEEE 산술에서 $\log 0 = -\infty$ 이고 $x < 0$ 일 때 NaN 이라고 정의하는 것은 자연스럽습니다. 만일 x 가 0 으로 언더플로우 된 매우 작은 음수라고 생각해봅시다. 그렇다면 0 에 부호가 있는 덕택에 x 는 음수가 될 것이므로, 로그 함수는 NaN 을 리턴하게 되겠지요. 만일 0 에 부호가 없다면, 로그 함수는 이 0 이 매우 작은 음수가 0 으로 언더플로우된건지 양수가 언더플로우 된 것인지 구별할 수 없기 대문에 그냥 $-\infty$ 를 리턴하게 됩니다. 다른 예제로 0 에서 불연속적인 signum 함수를 생각할 수 있는데요, 이 함수는 수의 부호를 리턴하는 함수 입니다.

부호 있는 0 을 사용하는 가장 흥미로운 예제로 복소수 연산을 들 수 있습니다. 간단한 예를 보자면, $\sqrt{1/z} = 1/(\sqrt{z})$ 를 생각해봅시다. 이 식은 $z \geq 0$ 일 때 자명하게 성립합니다. 하지만 $z = -1$ 일 때에는, 좌변의 경우 $\sqrt{1/(-1)} = \sqrt{-1} - i$ 가 되고, 우변의 경우 $1/(\sqrt{-1}) = 1/i = -i$ 가 되어서 성립하지 않게 되죠. 따라서 $\sqrt{1/z} \neq 1/(\sqrt{z})$ 가 되버리는 것입니다. 이러한 문제가 발생하게 된 근본적인 이유는, 제곱근 함수 자체가 다중 값 함수이기 (multi-valued function) 때문에 전체 복소 평면에서 연속이 되도록 값을 선택할 수 없기 때문입니다. 하지만, 모든 음의 실수를 포함하는 분기선 (branch) 을 제외하고 생각한다면 (이러한 것을 분기서법 (branch cut) 이라고 합니다) , 제곱근 함수를 연속으로 만들 수 있게 됩니다. 그럼 이제 $x > 0$ 일 때 $-x + i0$ 꼴의 남은 음의 실수들을 어떻게 처리해야 할지 문제가 생기게 됩니다. 여기서 부호 있는 0 은 이 문제를 해결하기 위한 완벽한 방법이 됩니다. $x + i(+0)$ 꼴의 수들은 한 개의 부호 $(i\sqrt{x})$ 를 가지고 분기선의 반대쪽에 있는 $x + i(-0)$ 꼴의 수는 부호가 $-i\sqrt{x}$ 가 됩니다. 사실, $\sqrt{}$ 를 계산하는 자연스러운 공식은 이러한 결과를 내게 되죠.

따라서 $\sqrt{1/z} = 1/(\sqrt{z})$ 로 다시 돌아가보면 만일 $z = 1 = -1 + i0$ 이라면,

$1/z = 1/(-1+i0) = [(-1-i0)]/[(-1+i0)(-1-i0)] = (-1-i0)/((-1)^2-0^2) = -1+i(-0)$

따라서 $\sqrt{1/z} = \sqrt{-1+i(-0)} = -i$ 이고, $1/(\sqrt{z}) = 1/i = -i$ 가 되서 등식이 성립합니다. 따라서 IEEE 산술은 모든 z 에 대한 항등식을 성립시켜줍니다. 좀더 복잡한 예들은 Kahan [1987] 을 참고하시기 바랍니다. 비록 $+0$ 과 -0 을 구분하는 일은 여러가지 장점이 있지만, 종종 혼동을 유발하기도 합니다. 예를 들어서 부호 있는 영은 $x = y \Leftrightarrow 1/x = 1/y$ 관계식을 만족시키지 않기 때문입니다. $(x = +0, y = -0)$ 하지만 IEEE 위원회는 부호 있는 0 을 사용하는데 얻는 이점이 그 단점을 훨씬 넘어섰다고 판단하였습니다.

2.2.4 정규화 되지 않는 수

예를 들어서 $\beta = 10, p = 3, e_{min} = -98$ 일 때 $x = 6.87 \times 10^{-97}, y = 6.81 \times 10^{-97}$ 들은 가능한 최소의 부동 소수점 수인 1.00×10^{-98} 보다 10 배나 큰 지극히 평범한 부동 소수점 수들 입니다. 그런데, 이들은 이상한 특징이 있는데, $x \neq y$ 임에도 불구하고 $x \ominus y = 0$ 이 된다는 것이지요. 이러한 문제가 발생하는 것은 $x - y = .06 \times 10^{-97} = 6.0 \times 10^{-99}$ 로 정규화 된 수로 표현하기에 너무나 작아서 결국 0 이 되버린다는 것입니다. 하지만 아래 관계를 유지하는 일은 매우 중요합니다.

$$x = y \Leftrightarrow x - y = 0 \tag{2.1}$$

만일 위 관계가 유지되지 않는다면 `if(x≠y then z = 1/(x-y)` 와 같은 매우 자명해 보이는 코드 조차 0 으로 나눗셈이 발생하여 오류가 발생할 수 있씁니다. 이러한 코드에서 발생하는 버그를 찾아내는 것은 매우 어려운 일이지요. 만일 식 2.1 가 명확하다면 매우 신뢰할 수 있는 부동 소수점 코드를 작성할 수 있겠지만, 대부분의 수에게만 참이고 일부 수에서는 거짓이라면, 아예 사용할 수 없겠지요.

IEEE 표준에서는 식 2.1 을 포함한 많은 유용한 관계식을 보장하기 위해 비정규화된 수(Denormalized numbers)라는 것을 도입하였습니다. 사실 이 부분은 표준에서도 가장 논쟁이 많이 되었던 부분이였고, 실제로 754 가 채택되기 전까지 많은 지연이 있었던 부분입니다. IEEE 표준에 호환이 된다고 주장하는 많은 고성능 하드웨어들은 비정규화 수들을 직접적으로 지원하지 않고, 비정규화가 발생하였을 대, IEEE 표준을 지원하도록 소프트웨어적인 측면에서 처리하도록 하여고 있습니다. [3] 비정규 수라는 아이디어를 최초로 제공한 것은 Goldberd [1967] 로 매우 간단한 아이디어입니다. 바로, 지수가 e_{min} 일 때, 가수는 정규화 될 필요가 없다는 것입니다. 따라서 $\beta = 10, p = 3, e_{min} = -98$ 일 때, 1.00×10^{-98} 은 더이상 가장 작은 부동 소수점 수가 아니게 됩니다. 왜냐하면

3) 이러한 사실은 표준에서 가장 문제가 되고 있는 부분이기도 합니다. 언더플로우를 자주 발생시키는 프로그램은 예외를 하드웨어보다 소프트웨어 상에서 처리 할 때 눈에 띄는 속도 저하를 느낄 수 있습니다.

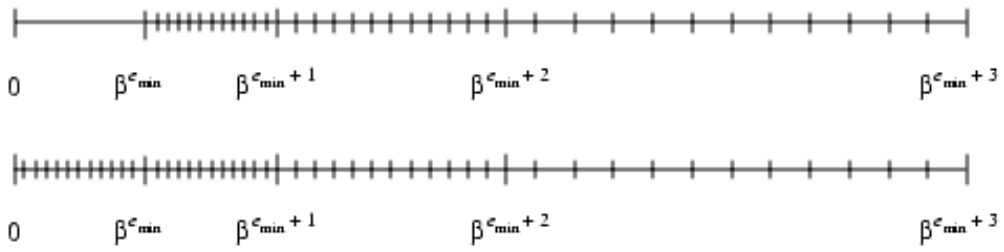

그림 2.1: 점진적 언더플로우를 사용할 때와 사용하지 않을 때

0.98×10^{-98} 역시 부동 소수점 수가 되기 때문입니다.

$\beta = 2$ 이고 숨겨진 비트가 사용되었을 대, 약간의 문제가 있는데 왜냐하면 지수가 e_{min} 여도 언제나 가수가 1.0 보다 크게 된다는 것입니다 (암묵적으로 맨 앞의 비트가 1 이니까). 그 해결책은 0 을 표현하는데 사용했던 것과 동일한 것으로 표 2.2 에 나와 있습니다. 지수 e_{min} 은 비정규화 수를 나타내는데 사용됩니다. 정확히 말하면, 만일 가수 비트들이 $b_1, b_2, ..., b_{p-1}$ 이고, 지수 값이 e 라 할 때, 만일 $e > e_{min} - 1$ 이면 수는 $1.b_1b_2...b_{p-1} \times 2^e$ 가 되고, 만일 $e = e_{min} - 1$ 이라면, 그 수는 $0.b_1b_2...b_{p-1} \times 2^{e+1}$ 로 처리됩니다. 이 때 $+1$ 이 필요한 이유는 비정규화 수들의 지수는 e_{min} 이지 $e_{min} - 1$ 이 아니기 때문입니다.

앞서 $x = 6.87 \times 10^{-97}, y = 6.81 \times 10^{-97}$ 의 예를 다시 생각해봅시다. 비정규화 수를 사용했다면 $x - y$ 는 0 이 되지 않고 그 대신에, 비정규화 수인 $.6 \times 10^{-98}$ 이 됩니다. 이러한 동작은 **점진적 언더플로우(gradual underflow)**라고 부릅니다. 그리고 점진적 언더플로우를 이용하면 식 2.1 을 증명하기 매우 쉽습니다.

그림 2.1 는 비정규화 수들을 보여주고 있습니다. 먼저 그림에서 윗줄에는 정규화 된 부동 소수점 수들을 보여주고 있는데, 0 과 가장 장근 정규화 된 수 간의 차이인 $1.0 \times \beta^{e_{min}}$ 을 볼 수 있습니다. 따라서 만일 계산된 결과가 이 사이 안에 떨어지게 된다면 바로 0 이 되버리게 된다는 것입니다. 하지만 아래줄을 보면, 비정규화 수를 사용하였을 때 어떠현 효과를 볼 수 있는지 알 수 있습니다. 그 '간극' 이 메꿔져있기 때문에, 계산 결과 크기가 $1.0 \times \beta^{e_{min}}$ 이 될지라도, 가장 가까운 비정규화 수가 되지 0 으로 떨어지지 않게 됩니다. 수직선에 비정규화 수들이 들어가게 된다면, 두 인접한 수 사이의 간격은 원래 정규화 수와 동일하게 작용합니다. 즉, 두 수 사이의 간격은 원래 정규화 수와 같거나, β 의 배수로 달라지게 되죠. 비정규화 수가 없다면, 그 간격이 $\beta^{-p+1}\beta^{e_{min}}$ 에서 $\beta^{e_{min}}$ 으로 급격히 차이가 나게 됩니다. 따라서 많은 알고리즘에서 언더플로우 문턱에서 많은 상대 오차를 보여주었던 정규화 된 수들의 경우, 점진적 언더플로우를 사용하게 되면 잘 작동하게 됩니다.

점진적 언더플로우를 사용하지 않을 경우 정규화된 입력들에 대해 간단한 식인 $x-y$ 와 같은 것 조차 앞에서 $x = 6.87 \times 10^{-97}, y = 6.81 \times 10^{-97}$ 의 예에서 보았듯이 매우 큰 상대 오차를 보여주게 됩니다. 아래 예제 [Demmel 1984] 에서 보여주듯이, 지워짐이 없더라도 큰 상대 오차는 발생할 수 있습니다. 아래와 같은 두 개의 복소수 $a+ib, c+id$

의 나눗셈을 살펴봅시다. 자명하게도, 그 식은

$$\frac{a+ib}{c+id} = \frac{ac+bd}{c^2+d^2} + i\frac{bc-ad}{c^2+d^2}$$

로 주어집니다. 앞에서도 이야기 하였지만 만일 분모 $c+id$ 의 절대값이 $\sqrt{\beta}\beta^{e_{max}/2}$ 보다 커진다면 실제 계산값이 범위안에 있더라도 오버플로우가 나게 됩니다. 따라서 위 식을 좀 더 제대로 계산하는 Smith 의 공식은

$$\frac{a+ib}{c+id} = \begin{cases} \frac{a+b(d/c)}{c+d(d/c)} + i\frac{b-a(d/c)}{c+d(d/c)} & \text{if } (|d| < |c|) \\ \frac{b+a(c/d)}{d+c(c/d)}i\frac{-a+b(c/d)}{d+c(c/d)} & \text{if } (|d| \geq |c|) \end{cases} \tag{2.2}$$

이 Smith 의 공식을 적용하게 되면 $(2\dot{1}0^{-98} + i10^{-98})/(4\dot{1}0^{-98} + i(2\dot{1}0^{-98}))$ 이 되어서 점진적 언더플로우인 0.5 를 내게 됩니다. 만일 점진적 언더플로우를 사용하지 않는다면 0.4 의 계산 결과는 0 으로 내림이 되어서 오차가 무려 100 ulps 에 달할 것입니다. 사실, 통상적인 경우 비정규화 수들은 오차의 한계를 $1.0 \times \beta^{e_{min}}$ 정도 까지 보장하게 됩니다.

2.3 예외, 플래그, 예외 처리기 (Exceptions, Flags, Trap handlers)

0 으로 나누기나 오버플로우와 같은 예외적인 상황들이 IEEE 산술 연산에서 발생할 경우, 디폴트 동작은 그냥 결과값을 전달하고 연산을 지속하는 것입니다. 보통 그 결과값들은 아마 0/0 이나 $\sqrt{-1}$ 의 경우 NaN 이겠고, 1/0 이나 오버플로우의 경우 ∞ 가 되겠지요. 이전 섹션들에서 이러한 디폴트 값들이 어떻게 사용될 수 있을지 이야기 하였습니다. 예외 상황들이 발생하였을 때 이러한 값을 리턴하는 것 뿐만이 아니라, 상태 플래그(status flag) 역시 설정이 됩니다. IEEE 표준에서는 유저들로 하여금 이렇나 상태 플래그의 값을 읽고 쓸 수 있도록 요구합니다. 이 플래그들은 한 번 설정이 되면, 명시적으로 클리어 되기 전 까지 값이 유지 됩니다. 플래그를 검사하는 것 만이 예컨대 ∞ 가 발생하였을 때 1/0 때문인지, 아니면 정말 오버플로우로 인한 무한대인지 구별할 수 있는 유일한 방법입니다.

때로는 예외상황이 발생하였을 때 작업을 지속하는 것이 적절하지 않을 수 도 있습니다. 무한대 에서 $x/(x^2+1)$ 과 같은 예를 들 수 있었는데요, 만일 $x > \sqrt{\beta}\beta^{e_{max}/2}$ 가 된다면 분모가 무한대가 되서 최종 계산값이 0 이 됩니다. 이전에도 이야기 했지만 위 식을 $1/(x+x^{-1})$ 로 다시 쓰게 된다면 위 문제를 해결할 수 있겠지만, 언제나 이렇게 식을 재구성 하는 것이 항상 답은 아닙니다. IEEE 표준은 **예외 처리기 (trap handler)** 의 도입을 가능케 하는 것을 강력하게 추천하여는데요, 따라서 예외가 발생하였을 때 플래그를 설정하는 것이 아니라 예외 핸들러가 호출되도록 하였습니다. 예외 핸들러에서 리턴된 값을 이용해서 작업을 적당히 처리하게 말이지요. 상태 플래그를 클리어 하던지 설정하던지 결정하는 것은 예외 핸들러의 몫입니다.

예외	예외 핸들러가 사용되지 않을 때	예외 핸들러에 전달되는 인자
오버플로우	$\pm\infty$ 나 $\pm x_{max}$	$round(x2^{-\alpha})$
언더플로우	0, $\pm 2^{e_{min}}$, 비정규화 수	$round(x2^{alpha})$
0 으로 나눠짐	$\pm\infty$	피연산자
올바르지 못한 작업	NaN	피연산자
부정확함	round(x)	round(x)

표 2.4: IEEE 754 의 예외들. 참고로 x 는 연산의 정확한 결과이고, 단정밀도의 경우 $\alpha = 192$, 배정밀도의 경우 $\alpha = 1546$ 입니다. 그리고 $x_{max} = 1.11...11 \times 2^{e_{max}}$ 입니다.

IEEE 표준은 예외를 5 개의 종류로 분류하였습니다. **오버플로우, 언더플로우, 0 으로 나눠짐, 올바르지 못한 작업(invalid operation), 그리고 부정확함(inexact)** 이죠. 각각의 예외 상황에 따라 독립적인 상태 플래그를 가지게 됩니다. 일단 앞서 말한 5개의 상태 플래그 중 3 개는 매우 명백합니다. 올바르지 못한 작업은 표 2.2.1 에서 나타난 상황들과 임의의 NaN 들 간의 비교를 의미합니다. 올바르지 못한 작업 예외가 발생하였을 때 디폴트로 NaN 이 리턴되지만, 그 역은 참이 아닙니다. 즉, 어떠한 연산에서 하나의 피연산자가 NaN 이라면, 그 결과는 NaN 이겠지만 그 연산이 표 2.2.1 의 조건을 만족하지 않는 이상 올바르지 못한 작업 예외는 발생하지 않게 됩니다.

부정확함 예외(inexact exception)는 부동 소수점 연산의 결과가 부정확 할 때 발생합니다. 예를 들어서 $\beta = 10, p = 3$ 인 시스템을 생각해봅시다. 그럼, $3.5 \otimes 4.2 = 14.7$ 은 정확하지만, $3.5 \otimes 4.3 = 15.0$ 은 부정확하므로 부정확함 예외를 발생시키게 됩니다. ($3.5\dot{4}.3 = 15.05$) 이진수를 십진수로 변환하기 에서 부정확함 예외를 사용하는 알고리즘에 대해 설명할 것입니다. 5 개 예외의 동작에 대해서는 표 2.3 에 정리해 놓았습니다.

인터럽트(interrupt) 란, 하드웨어 상에서 운영체제로 보내는 어떠한 신호를 뜻합니다. 주로 이 신호들은 운영체제에서 처리하는 작업을 중지하고 이 신호를 먼저 처리하기 때문에 인터럽트라 불립니다.

사실 부정확함 예외가 너무 많이 발생한다는 구현 상의 문제가 있습니다. 만일 부동소수점 하드웨어에 플래그가 있는 대신에 운영체제에 인터럽트를 보내는 방식으로 작동한다면, 너무나 많은 인터럽트가 운영체제로 전해져서 문제가 될 수 있기 때문입니다. 이러한 문제는 소프트웨어적으로 상태 플래그를 관리하는 방식으로 해결할 수 있습니다. 최초로 예외가 발생하였을 때, 이에 해당하는 소프트웨어 플래그를 설정한 뒤에, 부동 소수점 하드웨어로 하여금 그 클래스의 예외들을 무시하도록 설정 하면 되기 때문입니다. 따라서 그 뒤에 뒤따라 나오는 예외들은 모두 운영체제에 인터럽틀르 전달하지 않고 진행되며, 사용자가 상태 플래그를 리셋 하였을 때 다시 예외를 무시하지 않도록 설정하면 됩니다.

2.3.1 예외 처리기(Trap handler)

예외 처리기를 사용하는 한 가지 명백한 이유는 기존의 코드와 호환성을 위해서 입니다. 옛날 코들들의 경우, 예외가 발생하였을 때 작업이 중지지되어서 프로세스에 예외 처리기를 설치할 수 있을 것이라 가정하고 만들어져있습니다. 이는 특히 `do S until (x >= 100)` 과 같은 코드를 처리할 때 유용합니다. 왜냐하면 NaN 과 수를 $<, \leq, >, \geq, =$ (단 $\neq$ 는 아니다) 비교하는 일은 언제나 false 이므로 위 코드는 x

가 NaN 이 된다면 무한 루프를 돌게 때문이죠.

또한 예외 처리기는 오버플로우가 발생 할 수 있는 $\Pi_{i=1}x_i$ 와 같은 것을 처리할 때 유용하게 사용할 수 있습니다. 이에 대한 한 가지 해결책으로 로그를 사용하는 것인데, $\exp\sum\log x_i$ 를 대신 계산하는 것입니다. 하지만 이러한 방식으로 처리하였을 때 정밀도가 더 떨어지게 되고, $\Pi_{i=1}x_i$ 를 계산하는 것 보다 비용이 높아진다는 문제가 있습니다. 이 문제를 해결하는 다른 방법으로 오버/언더플로우의 개수를 세는 예외 처리기를 설치하여서 해결할 수 있습니다 [Sterbenz 1974].

그 방법은 다음과 같습니다. 0 으로 초기화 된 전역 카운터가 있고, 부분 곱 $p_k = \Pi_{i=1}^{k}x_i$ 라 정의한 다음에, 어떤 k 에서 오버플로우가 발생한다면 예외 처리기에서 카운터를 1 증가시키고, 지수를 다시 가능한 범위 안에 들어오도록 조정합니다. IEEE 754 단정밀도의 경우 $e_{max} = 127$ 이므로, 만일 $p_k = 1.45 \times 2^{130}$ 이 된다면, 오버플로우가 발생하게 되서 예외 처리기가 호출됩니다. 이 때, 예외 처리기는 지수를 단정밀도 범위에 포함되게 조정하여서 p_k 를 1.45×2^{-62} 로 조정합니다. (아래 참고) 비슷하게도, p_k 가 언더플로우가 나게 되면 카운터가 하나 줄어들며 음의 지수를 단정밀도 범위에 포함되도록 조정하게 됩니다. 만일 모든 곱셈이 수행된 이후에, 카운터가 0 이라면 그 최종 결과는 그대로 p_n 이 될 것입니다. 그런데, 카운터가 양수라면 그 결과값은 오버플로우 된 것이고 음수라면 언더플로우 된 것을 알 수 있게 됩니다. 만일 부분 곱에서 한 번이라도 범위를 초과하지 않았더라면 오류 처리기는 한 번도 호출되지 않기 때문에 추가적인 비용이 들지 않습니다. 만일 그 중간에 오버/언더 플로우가 발생하게 되었다고 해도, 그 계산 결과는 로그를 사용하였을 때 보다 더 정확하게 된다, 왜냐하면 p_k 는 p_{k-1} 로 부터 정확한 곱셈을 통해 계산되었기 때문입니다.

IEEE 754 에서는 예외 처리기가 호출될 때, 인자로 조정된 결과(wrapped around)를 전달하게 됩니다. 여기서 오버플로우의 조정된 결과라는 의미는, 그 결과값이 마치 무한대의 정밀도를 가지고 있는 것 처럼 계산된 다음에 2^{α} 로 나뉘어 진 후, 올바른 정밀도로 반올림 된다는 의미 입니다. 언더 플로우의 경우 그 반대로 2^{α} 가 곱해진 다음에, 동일하게 처리된다는 것이지요. 여기서 α 는 단정밀도의 경우 192, 배정밀도의 경우 1536 이 됩니다. 이 때문에 앞선 예에서 1.45×2^{130} 이 1.45×2^{-62} 가 된 이유지요.

2.3.2 반올림 방식들 (Rounding Modes)

IEEE 표준에서 반올림은 매 계산시에 항상 발생하게 됩니다. 기본값으로 반올림의 뜻은 가장 가까운 값으로 바꾼다는 의미 입니다. 표준에 따르면 3 개의 다른 반올림 방식을 지원하도록 하였는데요, 각각은 0 으로의 반올림, $+\infty$ 를 향한 반올림, 그리고 $-\infty$ 를 향한 반올림 입니다. 예를 들어서, 정수로의 변환 연산을 할 경우, $-\infty$ 를 향한 반올림은 마치 바닥 함수(floor function) 처럼 작동하고, $+\infty$ 를 향한 반올림은 마치 천장 함수(ceiling function) 처럼 작동하게 됩니다. 반올림 방식은 오버플로우에도 영향을 주게 되는데, 예를 들어서 0 으로의 반올림이나, $-\infty$ 를 향항 반올림이 작동중이라면, 양의 오버플로우가 발생하였을 때, $+\infty$ 가 아닌 가장 큰 표현 가능한 수를 리턴하도록 바뀌게 된다는 것입니다. 마찬가지로 음의 오버플로우가 발생하였을 때, 만일 $+\infty$ 나 $-\infty$ 를 향한 반올림은 쉽게 말해 내림(floor)이고, $+\infty$ 를 향한 반올림은 올림과(ceil) 동일합니다. 그리고 0 을 향한 반올림은 말 그대로 정수 부분만 취하는 것으로(truncate), 어떤 x 의 $sgn(x)\lfloor|y|\rfloor$ 와 동일합니다.

0 으로의 반올림이 작동중이라면 가장 큰 음의 수를 리턴하게 됩니다.

이 반올림 방식의 한 가지 응용 예로 구간 산술 연산(interval arithmetic) 을 들 수 있습니다. 예를 들어서 두 수 x, y 의 합은 구간 $[\underline{z}, \bar{z}]$ 안에 있는데, 여기서 $\underline{z}$ 는 $x \oplus y$ 가 $-\infty$ 로 반올림 된 것이고, 반대로 $\bar{z}$ 는 $x \oplus y$ 가 ∞ 로 반올림 된 것입니다. 만일 반올림 모드들을 사용하지 않는다면, 구간 산술은 보통 $\underline{z} = (x \oplus y)(1-\epsilon)$ 과 $\bar{z} = (x \oplus y)(1+\epsilon)$ (단, ϵ 은 머신 입실론) 으로 계산되며, 이는 구간의 크기를 꽤 크게 잡는 경향이 있습니다. 구간 연산의 결과는 항상 구간으로 표현되기 때문에, 구간 연산의 입력값 역시 구간으로 주어지는데요, 만일 두 개의 구간 $[\underline{x}, \bar{x}], [\underline{y}, \bar{y}]$ 을 더한다면, 그 결과는 $[\underline{z}, \bar{z}]$ 가 되며, 이 때 $\underline{z}$ 는 $-\infty$ 를 향한 반올림 방식에서 $\underline{x} \oplus \underline{y}$ 를 구한 것이고, $\bar{z}$ 는 $+\infty$ 를 향한 반올림 방식에서 $\bar{x} \oplus \bar{y}$ 를 구한 것입니다.

구간 산술 연산이란, 수학적으로 계산을 할 때, 반올림 오차와 수치적 오차를 고려하여 어떠한 구간 안에 포함시켜서 신뢰할 수 있는 계산 결과를 얻고자 하는 연산입니다. 예를 들어서, 사람의 키를 어림을 할 때 2.0 미터로 정확히 세는 것이 아니라, 그 키를 1.97 에서 2.03 미터 사이라고 구간을 정한 후 이를 가지고 연산하는 분야 입니다.

부동 소수점 연산이 구간 산술 연산으로 수행된다면, 최종 구간 안에는 반드시 연산의 정확한 결과가 포함됩니다. 구간의 크기가 매우 커진다면 (사실 대부분의 경우가 그렇습니다) 그 구간 연산의 결과는 별로 유용하지 못할 것입니다. 왜냐하면 참값이 구간 어디엔가 있는 것이기 때문이죠.

2.3.3 플래그 (Flag)

IEEE 표준에는 여러 개의 플래그 들과 모드들이 있습니다. 앞에서도 이야기 하였지만, 5 개의 예외(언더플로우, 오버플로우, 0 으로 나눔, 올바르지 못한 연산, 부정확함) 각각에 한 개의 상태 플래그가 할당 되어 있습니다. 또한 모드들의 경우 4 개로 앞에서도 이야기 하였듯이 $0, \infty, -\infty$ 를 향한 반올림과, 가장 가까운 수로의 반올림 (짝수 반올림) 말이지요. 이 섹션에서는 어떻게 하면 이 모드들과 플래그들을 잘 조합해서 사용할 수 있을지에 대해 다룰 것입니다. 좀 더 복잡한 예로 이진수를 십진수로 변환하기 을 참고해보시기 바랍니다.

예를 들어서 x^n 을 계산하는 루틴을 짠다고 생각해봅시다. 여기서 n 은 양의 정수인데, 여러분은 아래와 같이 간단히 짤 수 있을 것입니다.

```
PositivePower(x,n) {
 while (n is even) {
        x = x*x
        n = n/2
 }
 u = x
 while (true) {
        n = n/2
        if (n==0) return u
        x = x*x
        if (n is odd) u = u*x
 }
```

만일 $n < 0$ 이라면, x^n 을 계산하기에 좋은 방법이 `PositivePower(1/x, -n)` 을 호출하기 보다는 `1/PositivePower(x, -n)` 을 계산하는 것입니다. 왜냐하면 전자의 경우 이미 반올림 오차가 생기는 $1/x$ 를 n 번이나 곱해야 되기 때문이죠. 반면에 후자의 경우 맨 마지막 연산에서 한 번의 반올림 오차만이 생기는 것이기 때문에 좀 더 정확하다고 할 수 있습니다. 하지만 불행이도, 이 방법에는 약간의 문제가 있습니다. 만일 `PositivePower(x, -n)` 에서 언더플로우가 발생한다면, 언더플로우 예외 처리기가 작동하던지, 아니면 언더플로우 상태 플래그가 설정이 될 것입니다. 그런데 사실 이는 틀렸습니다. 왜냐하면 x^{-n} 이 언더플로우가 된다면, x^n 이 범위 안에 있을 수 있기 때문입니다. [4] 하지만 IEEE 표준에서는 사용자로 하여금 모든 플래그에 접근 가능하게 하므로 서브루틴에서 이 문제를 쉽게 고칠 수 있습니다. 먼저 오버플오우와 언더플로우 예외 가능 비트들을 꺼버리고, 오버플로우와 언더플로우 상태 비트들을 저장합니다. 그리고 `1/PositivePower(x, -n)` 을 계산한 다음에, 만일 두 상태 비트들 모두 설정되어 있지 않다면 이들의 예외 가능 비트들을 다시 켜버리면 됩니다. 만일 상태비트들 중 하나가 설정되어 있다면 플래그를 다시 복구한 뒤에, `PositivePower(1/x, -n)` 을 이용하여 계산을 해서 올바른 예외가 발생하도록 하면 됩니다.

플래그를 사용하는 다른 예제로, $\arccos x = 2\arctan\sqrt{\frac{1-x}{1+x}}$ 를 계산할 때가 있습니다. 만일 $\arctan(\infty)$ 가 정확히 $\pi/2$ 로 계산된다면, $\arccos -1$ 은 언제나 무한대 연산 덕택에 $2\arctan(\infty) = \pi$ 가 되겠지요. 하지만 여기에 약간의 문제가 있는데, $\arccos(-1)$ 이 전혀 예외적인 상황이 아님에도 불구하고 $\frac{1-x}{1+x}$ 는 0 으로 나눠짐 플래그를 설정해 버리기 때문입니다. 이 문제를 해결하는 방법은 직관적인데, 이 연산 이전에 0 으로 나눠짐 플래그의 값을 저장해 놓고 연산 후에 원래 플래그의 값으로 복구하면 됩니다.

---

4) 왜냐하면 $x < 1, n < 0$ 일 때 만일 x^{-n} 이 언더플로우 문턱인 $2^{e_{min}}$ 보다 겨우 몇 비트 차이가 난다면, $x^n = 2^{-e_{min}} < 2^{e_{max}}$ 가 될 수 있기 때문에 오버플로우가 나지 않을 수 있습니다. 참고로 모든 IEEE 표준에서 $-e_{min} < e_{max}$ 입니다

제 3 장

시스템 적 측면

컴퓨터 시스템을 설계함에 있어서 부동 소수점에 관련한 지식은 필수적이라 할 수 있습니다. 대부분의 컴퓨터 아키텍쳐에는 부동 소수점 연산이 포함되어 있고, 컴파일러들은 반드시 이러한 부동 소수점 명령을 생성할 수 있어야 하며, 운영 체제는 부동 소수점 예외가 발생하였을 때 무엇을 해야 할지 결정할 수 있어야만 합니다. 하지만 이러한 정보들은 주로 컴퓨터 디자이너들이 아닌 소프트웨어 제작자들에게 전달되지, 컴퓨터 시스템 디자이너들이 부동 소수점의 수치적 측면에 대해 정보를 얻을 기회가 많지 않습니다 . 아래 예제는 디자인 측면에서 타당해 보이는 BASIC 코드가 어떻게 예측하지 못한 동작을 할 수 있는지 살펴보도록 합시다.

```
q = 3.0/7.0
if q = 3.0/7.0 then print "Equal":
        else print "Not Equal"
```

놀랍게도 IBM PC 에서 Borland 사의 Turbo Basic 을 이용해서 컴파일 하면 놀랍게도, "Not Equal" 을 출력하게 됩니다. 이 예제는 다음 명령어 집합(instruction sets)에서 살펴보도록 합시다.

그런데, 어떤 사람들은 위와 같은 코드에서 발생하는 문제가 바로 부동 소수점을 어떠한 오차 범위 E 를 가지고 비교하는 것이 아니라 직접적으로 비교함으로써 발생하는 오류라고 생각합니다. 하지만 이러한 처방은 몇 가지 궁금증을 불러 일으키는데, 과연 그렇다면 E 의 값으로 얼마를 써야 할까요? 만일 $x < 0$ 이고 $y > 0$ 인데 E 범위 안에 있다면 두 수의 부호 자체가 다르지만 같다고 해야 할까요. 게다가 $ab \Leftrightarrow |a - b| < E$ 는 동치가 아닌데, 왜냐하면 과 $abandbc$ 라고 해서 ac 를 보장하지는 않기 때문입니다.

3.1 명령어 집합(instruction sets)

하드웨어에서 지원하는 명령어들을 모아놓은 것을 명령어 집합이라고 부릅니다.

어떤 알고리즘에서 계산을 할 때 계산 중간에 특정 계산들은 높은 정밀도로 계산해야 하는 경우가 종종 있습니다. 예를 들어서, 이차방정식의 근의 공식 $(-b \pm \sqrt{b^2 - 4ac})/2a$ 을 계산하는 것을 생각해봅시다. 이전에도 이야기 하였지만, $b^2 \approx 4ac$ 일 경우, 반올림 오차로 인하여 제곱근 연산시 자리수의 절반 이상을 신뢰할 수 없게 만들 수 있습니다.

하지만 $b^2 - 4ac$ 부분만 단정밀도로 아닌 배정밀도로 계산을 하게 된다면, 배정밀도의 절반 자리수가 부정확하더라도, 단정밀도로 변환시에 전체 자리수를 신뢰할 수 있게 되는 것입니다.

만일 두 개의 단정밀도 값을 입력 받아서 곱셈을 한 후 배정밀도로 결과를 돌려주는 명령어가 있다면 a, b, c 가 단정밀도 일 때 $b^2 - 4ac$ 를 배정밀도로 계산하는 일은 어렵지 않을 것입니다. 정확히 반올림되는 두 개의 p 자리 수의 곱을 계산하기 위해서, 어차피 곱셈기(multiplier)는 대략 $2p$ 자리의 결과값을 생성하기 때문에 하드웨어로 하여금 두 개의 단정밀도 값을 인자로 받아서 결과값을 배정밀도로 계산하는 것은 결과값을 단정밀도로 계산하는 것 보다 약간의 비용이 더 들 뿐더러, 배정밀도 곱셈기를 이용하는 것보다는 훨씬 효과적입니다. 그럼에도 불구하고, 대부분의 명령어 집합은 피연산자와 같은 타입으로 결과값을 반환하는 명령만을 지원하는 경우가 많습니다.

만일 두 개의 단정밀도 피연산자를 가지고 배정밀도 결과값을 반환하는 명령이 오직 근의 공식 뿐이라면, 이 새로운 명령을 명령어 집합에 포함시키는 일이 별로 효과적이지 않을 수 도 있습니다. 하지만, 이 명령어는 그 외 수 많은 용도로 사용 가능합니다. 예를 들어 다음과 같은 연립 일차방정식들을 푸는 시스템을 생각해봅시다.

$$\begin{aligned}
&a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\
&a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\
&\cdots \\
&a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n
\end{aligned}$$

이는 행렬을 이용해서 $Ax = b$ 꼴로 다시 쓸 수 있는데, 여기서

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ & \cdots & & \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$

로 쓸 수 있습니다. 이제, 위 방정식의 한 해 $x^{(1)}$ 를 가우시안 소거법(Gaussian elimination)과 같은 방법을 사용해서 구했다고 생각해봅시다. 그럼, 그 해의 정확도를 높일 수 있는 간단한 방법이 하나 있는데요, 바로 반복 향상법(iterative improvement)라고 부르는 방법입니다. 먼저

$$\zeta = Ax^{(1)} - b \tag{3.1}$$

를 계산합니다. 그리고, 아래와 같은 시스템의 해를 구합니다.

$$Ay = \zeta \tag{3.2}$$

만일 $x^{(1)}$ 이 정확한 해였다면, ζ 는 영벡터였을 것이고, 따라서 y 도 영벡터였을 것입니다. 하지만, 일반적으로 ζ 와 y 를 계산하면서 반올림 오차가 생기기 때문에

$Ay \approx \zeta \approx Ax^{(1)} - b = A(x^{(1)} - x)$ 가 됩니다. (여기서 x 는 알려지지 않은 참값) 그렇다면 $y \approx x^{(1)} - x$ 가 되기 때문에, 해의 좀더 향상된 근사치로

$$x^{(2)} = x^{(1)} - y \tag{3.3}$$

를 생각할 수 있습니다. 여기서 $x^{(1)}$ 을 $x^{(2)}$ 로 바꾸고, $x^{(2)}$ 를 $x^{(3)}$ 으로 바꾸는 식으로 해서 세 단계 3.1, 3.2, 3.3 를 계속 반복적으로 수행할 수 있습니다. 이 때, 구하게 된 $x^{(i+1)}$ 은 언제나 $x^{(i)}$ 보다 정확하게 되는데, 자세한 내용은 [Golub and Van Loan 1989] 를 참조하시기 바랍니다.

반복 향상법을 사용할 때, ζ 벡터의 원소들은 모두 근접한 부동소수점 수들의 차이로 구성되어 있기 때문에, 나쁜 지워짐이 발생할 수 있습니다. 따라서 $\zeta = Ax^{(1)} - b$ 가 배정밀도로 계산되지 않는 한 큰 효과를 볼 수 없겠지요. 이는 역시, 두 개의 단정밀도 수 $(A, x^{(1)})$ 들의 곱셈을 배정밀도 결과값으로 반환해야 하는 경우 입니다.

정리하자면, 두 개의 부동 소수점 수를 곱한 다음 그 결과를 피연산자의 두 배 정밀도로 반환하는 명령을 부동 소수점 명령어 집합에 추가하는 것은 매우 효율적일 것입니다. 이러한 명령어를 컴파일러에 어떻게 구현하는냐는 다음 섹션에서 다루고 있습니다.

3.1.1 언어와 컴파일러 (Languages and Compilers)

컴파일러와 부동 소수점과의 관계는 Farnum [1988] 의 논문에서 잘 찾아볼 수 있고, 이 섹션에서 다루는 대부분의 내용은 그 논문에서 따온 것입니다.

모호함 (Ambiguity)

이상적으로 컴퓨터 언어는 언어의 모든 요소들을 정확하게 정의하여서 프로그램의 모든 문장들을 엄밀하게 증명할 수 있어야만 합니다. 이러한 사실은 언어의 정수 부분에서 참인 가운데, 부동 소수점의 경우 엄밀하게 정의되어 있지 않은 경우가 많습니다. 이는 많은 언어 설계자들이 부동 소수점 부분을 정의할 때, 부동 소수점 연산은 언제나 반올림 오차를 수반하기 때문에 엄밀하게 증명하는 것은 불가능 한 것이라 판단한 것으로 생각됩니다. 하지만 우리는 이전 섹션들에서 이러한 논리는 틀렸다는 것을 보여왔습니다. 모호함 (Ambiguity) 에서는 언어 정의들에서 공통적으로 모호하게 다루어지는 정의 부분에 대해 논의할 것이고, 이들을 어떻게 처리해야 하는지도 제안할 것입니다.

놀랍게도, 몇몇 언어에서는 x 가 부동 소수점 변수 일 때 (예를 들어 값이 3.0/10.0 이라고 합시다), 프로그램의 모든 부분에서 `10.0 * x` 가 같은 값을 가지도록 명확히 보장하지 않는다는 점에 있습니다. 예를 들어서 Brown 의 공리를 기반으로 한 Ada 라는 언어의 경우, 부동 소수점 연산은 오직 Brown 의 공리만을 만족시킬 필요가 있는 것처럼 작동하는데, 따라서 많은 가능한 값들 중에 임의로 그들 중 하나의 값을 가지면 된다고 설계되어 있습니다. 이는 모든 부동 소수점 연산이 정확하게 정의되어 있는 IEEE

모델과 정 반대입니다. IEEE 모델에서는 우리는 `3.0/10.0)*10.0` 이 3 과 동일하다는 사실을 (정리 1.5.3) 증명할 수 있습니다. 하지만 Brown 의 모델은 그렇지 않습니다.

대부분의 언어의 정의에서 나타나는 모호함으로 오버플로우와 언더플로우 그리고 다른 예외에서 찾을 수 있습니다. IEEE 표준에서는 명확하게 이들 각각의 예외가 발생하였을 때의 처리를 규정해 놓았기 때문에, 언어측면에서 이 표준을 모델로 사용한다면 모호함을 해결할 수 있습니다.

또 다른 모호한 부분으로, 괄호를 해석하는데 관련이 있습니다. 반올림 오차로 인하여, 부동 소수점 수의 경우 결합 법칙이 성립하지 않을 수 있습니다. 예를 들어서 `(x+y)+z` 는 `x+(y+z)` 와 전혀 다를 수 있습니다. (예를 들어서 $x = 10^{30}, y = -10^{30}, z = 1$) 따라서 괄호를 정확히 지키는 것의 중요성은 정말 아무리 강조해도 지나칠 수 없습니다. 심지어 이전에 정리 1.4.1, 1.4.2, 1.5.2 에서 보았듯이, 이들은 모두 괄호에 기반을 하고 있습니다. 예를 들어 정리 1.5.2 에서 식 $x_h = mx - (mx - x)$ 는 만일 괄호가 없었더라면 $x_h = x$ 가 되어서 전체 알고리즘을 망가뜨렸습니다. 언어 측면에서 괄호를 명확히 지키지 않는다면, 부동 소수점 계산은 무용지물이 될 것입니다.

많은 컴퓨터 언어에서 부분식 계산은 부정확하게 정의되어 있습니다. 예를 들어서 ds 는 배정밀도이고 x, y 는 단정밀도라고 해봅시다. 그렇다면 `ds + x*y` 는 단정밀도로 계산되어야 할가요, 아니면 배정밀도로 계산되어야 할까요. 다른 예제로, `x + m/n` 에서 m, n 이 모두 정수 일 때, 나눗셈을 정수 나눗셈으로 해야 할까요, 부동 소수점 나눗셈으로 해야 할까요. 이러한 문제에 대해 해결하는 방으로 두 가지 방법이 있는데, 이 두 방법 모두 완벽히 만족스럽지는 않습니다. 첫 번째 방법은, 어떠한 식에서 모든 변수들이 같은 타입이 되도록 강요시키는 것입니다. 이는 매우 간단한 해결책이지만 몇 가지 문제점이 있습니다. 예를 들어서 Pascal 과 같은 언어에서는 부분 범위 타입 (subrange type) 이란 것이 있어서, 부분 범위 변수들과 정수 변수들을 혼합해서 사용하는 것을 허용하고 있습니다. 따라서, 단정밀도와 배정밀도 변수들을 혼합하는 것을 제한한다는 것은 이상한 일이지요. 다른 문제로, 상수들을 다룰 때, 예를 들어서 `0.1*x` 를 계산할 때 대부분의 언어에서는 0.1 을 단정밀도 상수로 해석합니다. 만일 프로그래머가 모든 부동 소수점 변수의 정의를 단정밀도에서 배정밀도로 바꾼다고 하면, 0.1 은 계속 단정밀도 상수로 취급되기 때문에 이는 컴파일 오류를 발생시키게 됩니다. 따라서 프로그래머들은 이를 고치기 위해 모든 부동 소수점 상수들을 찾아 나서야 하겠지요.

두번째 방법으로는 혼합 연산을 허용하는 대신에, 부분 식들을 계산할 때 반드시 규칙들을 정의하는 것입니다. 예를 들어 원래 C 언어 정의에 따르면 모든 부동 소수점 수들은 배정밀도로 계산되게 요구하였습니다 [Kernighan and Ritchie 1978]. 이를 통해서 이 섹션 맨 처음에 나온 예제같은 문제점들을 해결할 수 있었습니다. `3.0/7.0` 과 같은 식들은 모두 배정밀도로 변환되서 계산되는데 q 는 단정밀도 이기 때문에 그 몫은 다시 단정밀도로 반올림 되서 저장됩니다. 그런데, 3/7 는 이진수로 무한히 반복되는 무한소수이기 때문에, 배정밀도로 계산된 값과, 단정밀도로 저장된 값에는 차이가 생기게 됩니다. 따라서 `q == 3/7` 과 같은 문장은 성립하지 않게 됩니다. 결과적으로, 모든 식을 가능한 높은 정밀도로 계산하는 것은 좋은 해결책이 아닐 수 있습니다.

위 문제에 대한 해답을 제시해 줄 수 있는 예로 내적(inner product) 계산이 있습니

다. 만일 천 개가 넘는 항들의 내적 계산을 수행한다면, 합에서 발생하는 반올림 오차는 상당히 클 수 있습니다. 이러한 반올림 오차를 줄이는 방법으로 이 합들을 배정밀도로 수행하는데 있습니다. (이는 최적화기 에서 이야기 할 것입니다) 만일 `d` 가 배정밀도 변수이고, `x[]` 와 `y[]` 가 단정밀도 배열들이라면, 내적 계산 루프는 `d = d+x[i]*y[i]` 처럼 생겼겠지요. 만일 곱셈이 단정밀도로 계산된다면, 배정밀도로 덧셈을 해서 얻는 이점이 줄어들 것입니다. 왜냐하면 계산된 단정밀도의 곱은 배정밀도 변수에 더해지기 직전에 단정밀도로 잘려나가기 때문입니다.

위 두 예제를 아우르는 하나의 규칙은 바로, 그 식에서 가장 높은 정밀도의 변수로 계산을 한다는 것입니다. 그렇게 된다면 `q == 3.0/7.0` 은 그 식에서 가장 높은 정밀도가 단정밀도 이므로 [1], 단정밀도로 계산되어 식이 참이 되겠고, 마찬가지로 `d = d+x[i]*y[i]` 는 배정밀도로 계산되어 배정밀도 덧셈의 모든 효과를 누릴 수 있게 됩니다. 하지만, 이 규칙은 모든 경우를 깔끔하게 처리하기에는 너무 간단합니다. 예를 들어서 `dx, dy` 가 배정밀도 변수라면, 식 `y = x + single(dx-dy)` 는 배정밀도 변수를 가지고 있지만, 합을 배정밀도로 하는 것은 의미가 없게 됩니다. 왜냐하면 두 피연산자 모두 단정밀도 이기 때문이죠.

조금 더 상세한 규칙은 다음과 같습니다. 일단, 각각의 연산에 잠정적인 정밀도를 부여합니다. 이 잠정적인 정밀도는 피연산자들의 최대 정밀도로 구성됩니다. 이 잠정적인 정밀도는 식 트리(expression tree) 의 잎 부터 뿌리 까지 모든 부분에 해당되게 됩니다. 그 다음으로, 뿌리에서 잎으로 쭉 읽어들이게 되는데, 각각의 연산에 잠정적 정밀도와 부모로 부터 기대되는 정밀도를 중 최대값을 할당하게 됩니다. `q == 3.0/7.0` 의 경우, 모든 잎이 단정밀도 이기 때문에, 모든 작업은 단정밀도에서 작동하게 됩니다. `d = d+x[i]*y[i]` 의 경우, 곱셈 연산에서 잠정적 정밀도는 단정밀도 이지만 그 다음을 읽어들일 때 부모 연산에서 피연산자를 배정밀도로 기대하기 때문에 배정밀도가 되버리게 됩니다. 그리고 마지막으로 `y=x+single(dx-dy)` 의 경우, 덧셈이 단정밀도로 수행되게 됩니다. Farnum [1988] 은 이 알고리즘을 구현하는데 큰 어려움이 없음을 보였습니다.

식을 트리로 분해해서 생각하는 것입니다. 자세한 것은 `http://en.wikipedia.org/wiki/Binary_expression_tree` 을 참고하세요

이 규칙의 단점은로는, 부분식을 계산할 때 정밀도가 이 식이 포함되어 있는 전체식에 의존된다는 것입니다. 이러한 사실은 몇 가지 짜증나는 문제점을 야기할 수 있는데, 예를 들어서 여러분이 프로그램을 디버깅하는데 중간식의 값을 알고싶다고 합시다. 하짐나 여러분은 단순히 디버거에 그 부분식을 쳐서 그 값을 계산해달라고 요청할 수 없습니다. 왜냐하면 부분식의 값은 그 식이 포함되어 있는 전체식에 의존하기 때문에 그 값만 따로 뽑아 올 수 없게 되는 것입니다. 마지막으로 부분식에 대해 한 가지 코멘트 해보자면, 십진수 상수를 이진수로 변환하는 과정에서 부분식 계산 규칙이 십진수 상수를 해석하는데 중요한 역할을 한다는 사실입니다. 이는 0.1 과 같은 이진수로 정확히 표현할 수 없는 (무한 소수) 들을 처리하는데 중요한 사실입니다.

언어의 또다른 가능한 모호함은 지수함수를 내장 연산으로 계산하면서 발생합니다.

1) 여기서 3.0 은 흔히 단정밀도 상수임을 가정합니다. 참고로 배정밀도 상수의 경우 3.0D0 로 표현됩니다.

다른 기초 산술 연산과는 달리, 지수의 계산 결과는 언제나 명백한 것이 아닙니다 [Kahan and Coonen 1982]. 만일 `**` 가 지수 연산자였다면, `(-3)**3` 의 값은 자명히 -27 이겠지요. 하지만 `(-3.0)**3.0` 이면 이야기가 달라집니다. 만일 `**` 연산자가 정수 지수만 받는다면, `(-3.0)**3` 을 계산해서 $-3.0^3 = -27$ 을 정상적으로 출력할 것입니다. 반면에 실수 지수를 위해서 $x^y = e^{y \log x}$ 를 이용에 계산을 하게 된다면, 로그 함수에 따라 그결과가 NaN 이 될 수 도 있습니다. ($x < 0$ 일 때 $\log(x) = NaN$) 이 될 수도 있는 것입니다. 하지만 FORTRAN 의 `CLOG` 함수를 이용하게 된다면, 그 답은 정확히 -27 로 나올 것인데, 왜냐하면 ANSI FORTRAN 표준에 따르면 `CLOG(-3.0)` 이 $i\pi + \log 3$ 이 되도록 정의하고 있기 때문입니다. Ada 프로그래밍 언어에서는 아예 지수 부분을 오직 정수 만 받도록 하여서 이러한 문제를 피해가고 있으며, ANSI FORTRAN 에서는 음의 밑수의 실수 지수를 계산하는 것을 제한하고 있습니다.

사실 FORTRAN 표준에 따르면

> 그 연산 결과가 정확히 수학적으로 정의되지 않은 연산들은 모두 제한된다 (Any arithmetic operation whose result is not mathematically defined is prohibited ..)

불행이도, IEEE 표준에 $\pm\infty$ 를 도입하였기 때문에 **수학적으로 엄밀하게 정의되지 않는다** 라는 말은 더이상 명확하지 않게 되었습니다. 물론, 무한대 에서 정의한 한 가지 방법으로 사용할 수 도 있을 것입니다. 예를 들어서 a^b 의 값을 정의할 때, 상수가 아닌 해석함수 f, g 가 $x \to 0$ 일 때 $f(x) \to a, g(x) \to b$ 라고 합시. 그렇다면, $f(x)^{g(x)}$ 역시 같은 극한 값을 가지며, 그 값은 a^b 가 될 것입니다. 이러한 정의에 따르면 $2^\infty = \infty$ 가 된다는 것은 자명해 보입니다. 하지만 1.0^∞ 의 경우, $f(x) = 1$ 이고 $g(x) = 1/x$ 라 하면 그 극한값은 1 이 되겠지만, $f(x) = 1 - x$ 이고 $g(x) = 1/x$ 라 하면 극한값은 e^{-1} 이 됩니다. 따라서 1.0^∞ 는 NaN 이 되어야 합니다. 0^0 의 경우, $f(x)^{g(x)} = e^{g(x) \log f(x)}$ 라 할 때 f, g 모두 0 에서 값이 0 인 해석함수로 정의한다면 (예를 들어 $f(x) = a_1x^1 + a_2x^2 + ..., g(x) = b_1x^1 + b_2x^2 + ...$ 라 하면 된다) $\lim_{x\to 0} g(x) \log f(x) = \lim_{x\to 0} x \log(x(a_1 + a_2x + ...)) = \lim_{x\to 0} x \log(a_1x) = 0$ 이 됩니다. 따라서, 모든 f, g 에 대해 $f(x)^{g(x)} \to e^0 = 1$ 가 되므로 $0^0 = 1$ 이 됩니다. [2] 이러한 정의를 사용하면 지수함수의 임의의 인자들에 대해서도 모호하지 않게 정의할 수 있고 특히 `(-3.0) ** 3.0` 역시 -27 이 되도록 정의할 수 있게 됩니다.

3.1.2 IEEE 표준

이전에 에서 IEEE 표준의 여러 특징들에 대해 설명하였습니다. 하지만 IEEE 표준에서는 이러한 기능들에 프로그래밍 언어에서 어떠한 방식으로 접근을 해야하는지에 대해서는 아무런 설명도 하지 않습니다. 이 때문에, IEEE 표준을 지원하는 부동 소수점 하드웨어와 C, Pascal, FORTRAN 과 같은 언어들간의 불일치가 생기게 됩니다. IEEE

2) $0^0 = 1$ 이라는 결과는 f 가 상수 함수가 아니라는 조건이 필요합니다. 만일 이 조건을 없앤다면, f 를 값이 0 인 상수 함수로 두고 $\lim_{x\to 0} f^g$ 를 계산하면, 0^0 은 NaN 이 됩니다

서브루틴(subroutine) 이란 쉽게 말해 우리가 생각하는 함수와 거의 동일하다.

의 일부 기능들은 여러 서브루틴 라이브러리들을 통해 접근 가능합니다. 예를 들어서 IEEE 표준은 제곱근 계산이 정확히 반올림 되도록 요구하는데, 따라서 제곱근 함수는 하드웨어에 측면에서 직접적으로 지원이 됩니다. 이러한 기능들은 제곱근 루틴 라이브러리를 통해서 쉽게 접근할 수 있습니다. 하지만, IEEE 표준의 몇 가지 다른 기능들은 서브루틴으로 쉽게 구현되기 어렵습니다. 예를 들어서 대부분의 컴퓨터 언어에서는 두 개의 부동 소수점 타입을 지원하는데요, 반면에 IEEE 표준에서는 4 개의 서로 다른 정밀도를 지원하도록 요구합니다 (특히 단정밀도, 확장 단정밀도, 배정밀도, 확장 배정밀도 들을 말이지요). 무한대 또한 다른 예가 될 수 있습니다. 서브루틴 차원에서 $\pm\infty$ 를 나타내는 상수를 지원할 수 있습니다. 하지만, 상수 변수의 초기화 함수 들과 같이, 상수 식을 필요로 하는 곳에서는 이러한 서브루틴을 사용할 수 없게 됩니다.

좀 더 미묘한 상황으로, 연산시에 상태(반올림 모드, 예외 가능 비트, 예외 처리기, 예외 플래그 등등)를 조작하는 경우가 있습니다. 이러한 상태를 읽고 또 쓰기 위한 한 가지 방법으로 서브루틴을 제공하는 방법이 있습니다. 게다가, 새로운 값을 설정하고, 이전의 값을 반환하는 루틴은 매우 유용합니다. 이전에 플래그(Flag)의 예시들에서 다루어 왔듯이, IEEE 상태를 조작해서 이용하는 대부분의 경우는 단순히 그 서브루틴 범위 안에서만 해당합니다. 따라서, 프로그래머는 그 루틴 밖으로 벗어나게 된다면, 원래의 상태로 반드시 복구를 해주어야 할 의무가 생기게 됩니다. 따라서 특정 루틴 안에서만 상태를 정확하게 조절하고 또 루틴이 끝나면 이를 복구할 수 있도록 언어 차원에서 지원하는 것은 매우 유용한 일입니다. Modula-3 언어는 이러한 기능을 구현한 한 언어 입니다 [Nelson 1991].

IEEE 표준을 언어에서 구현할 때 신경을 써야할 몇 가지 부분들이 있습니다. 예를 들어 모든 x 에 대해서, $x - x = +0$ 이므로, $(+0) - (+0) = +0$ 이 됩니다. [3]하지만 $-(+0) = -0$ 이므로, $-x$ 는 $0 - x$ 로 정의되면 안됩니다. 또한 NaN 들을 도입하는 것은 꽤나 혼동될 수 있는데 왜냐하면 NaN 은 그 어떤 수와도 (심지어 NaN 들 끼리도) 같을 수 가 없기 때문에 $x = x$ 라는 문장도 언제나 참이 아닙니다. 사실, IEEE 표준에서 $x \neq x$ 는 만일 `Isnan` 함수를 사용할 수 없을 때, x 가 NaN 인지를 테스트 하는 가장 효과적인 방법 중 하나입니다. 게다가 NaN 는 다른 수와의 대소 관계를 정할 수 없기 때문에 , $x \leq y$ 가 $x > y$ 가 아니다로 정의할 수 없게 됩니다. 따라서 NaN 들을 때문에 부동 소수점 수들의 일부분 만이 정렬될 수 있기 때문에 프로그래머로 하여금 비교 문제를 쉽게 처리하기 위해 `compare` 함수는 $<, =, >$ 혹은 *unordered* 를 리턴하게 됩니다.

마치 허수들 처럼 대소 비교가 불가능 합니다.

비론 IEEE 표준에서 기초 부동 소수점 연산에 인자로 NaN 이 들어갔다면 그 연산 결과로 NaN 이 반환하도록 정의하였지만 혼합 연산에서 이는 항상 만족스러운 결과를 내는 것은 아닙니다. 예를 들어서 그래프를 화면에 그리기 위해 몇 배율로 그릴 지 연산을 할 수 있을 것입니다. 이를 위해서는 그래프의 가능한 가장 최대의 값이 반드시 계산되어야 하는데, 이 때 계산 결과에 NaN 이 있더라도 이를 단순히 무시해버리면 되기 때문입니다.

3) 물론 이 말은 $-\infty$ 로의 반올림을 하지 않았을 대 이야기 입니다. 그 경우에는 $x - x = -0$ 이 됩니다.

마지막으로 반올림 또한 문제가 될 수 있습니다. IEEE 표준에서는 아주 세세하게 반올림을 정의하였는데요, 어떻게 반올림을 할 지에 대해서는 현재의 반올림 형식에 의해 결정됩니다. 이는 때때로 타입 변환 시에 암묵적으로 일어나는 반올림이나, 몇몇 언어들의 `round` 함수와 충돌을 일으킬 때가 있습니다. 이 말은, IEEE 반올림을 사용하고자 하는 프로그램들이 언어적 차원에서 지원하는 반올림 밖에 사용하지 못해 IEEE 장치들 간에 이를 이식하는데 문제가 발생할 수 있습니다.

3.2 최적화기

컴파일러에 관련한 문서들을 보면, 많은 경우 부동 소수점에 관련한 내용을 무시하는 경향이 있습니다. 예를 들어 Aho et al. [1986] 에서는 `x/2.0` 을 `x *0.5` 로 치환하는 것에 대해 이야기 하였는데, 이는 마치 독자로 하여금 `x/10.0` 역시 `0.1 * x` 로 바뀌어야 한다고 생각하도록 만들었습니다. 하지만 이 둘은 이진 기계에서 전혀 다른 의미를 지니게 되는데 왜냐하면 0.1 은 이진수로 정확히 표현될 수 없기 때문입니다. 또한 이 책에서 `x*y-x*z` 를 `x*(y-z)` 로 치환하는 것을 이야기 하였는데, 우리가 앞에서도 보았지만, 만일 $y \approx z$ 라면 두 개의 값은 꽤 달라 질 수 있기 때문입니다. 어찌 되었던 간에, 컴퓨터 언어 표준에서는 반드시 괄호를 보존해야 된다는 점을 명시해야 합니다. 왜냐하면 앞에서도 이야기 하였듯이 `(x+y)+z` 와 `x+(y+z)` 은 완전히 다른 값이 될 수 있기 때문이지요. 괄호를 보존하는 것에 관련해서 아래와 같은 코드에서 문제점을 하나 볼 수 있습니다.

```
eps = 1;
do eps = 0.6*eps; while (eps + 1 > 1);
```

이 코드는 머신 입실론의 근사치를 구하도록 제작된 것입니다. 만일 컴파일러의 최적화기가 $eps + 1 > 1 \Leftrightarrow eps > 0$ 이라는 것을 발견하고 이를 수정한다면 위 프로그램은 완전히 달라지게 됩니다. 위 코드를 변경하게 되면, x 에 $1 \oplus x$ 를 해도 1 ($x \approx e \approx \beta^{-p}$) 보다 커지게 하는 가장 작은 수 x 를 찾는 것이 아니라, $x/2$ 가 반올림 되어서 0 ($x \approx \beta^{e_{min}}$) 이 되는 가장 큰 x 를 찾는 루틴으로 바뀌게 되는 것입니다. 이러한 종류의 최적화를 막는 것은 매우 중요해서, 이러한 최적화 하나 때문에 알고리즘의 의미 자체가 달라질 수 있습니다.

많은 경우 0 과 비교하는 것이 더 빠릅니다. 대부분의 컴퓨터 하드웨어 들에는 0 과 비교하는 명령어들이 따로 만들어져 있습니다.

많은 문제들 - 예를 들어 수치 적분을 한다던지, 미분 방정식의 수치 해를 구한다든지 에서 다수의 항의 덧셈을 필요로 하는 경우가 많습니다. 이 때 각각의 덧셈에서 최대 .5 ulp 의 오차가 발생하기 때문에 수천개의 항을 포함하는 덧셈 과정에서 꽤 많은 비트의 반올림 오차가 발생하게 됩니다. 이러한 문제를 해결하는 가장 간단한 방법은 부분 합을 원래 변수의 두 배 크기의 정밀도 변수에 저장하며, 각각의 덧셈을 두 배의 정밀도로 수행하면 된다는 것입니다. 만일 연산이 원래 단정밀도에서 수행된다면 덧셈을 배정밀도로 올려서 수행하는 것은 그리 어렵지 않을 것입니다. 반면에 연산 자체가 배정밀도 변수들이였다면, 이에 두 배가 되는 정밀도를 사용하는 것은 꽤 골치아픈 문제입니다. 하지만, 이러한 덧셈의 정확도를 엄청나게 향상시킨 훨씬 좋은 방법이 있습니다.

Theorem 3.2.1. *Kahan 의 합 공식 (Kahan Summation Formula).* 만일 $\sum_{j=1}^{N} x_j$ 가 다음의 알고리즘의로 계산되었다고 하자.

```
S = X[1];
C = 0;
for j = 2 to N {
        Y = X[j] - C;
        T = S + Y;
        C = (T - S) - Y;
        S = T
}
```

그렇다면 최종 계산된 합인 S 는 $\sum_{x_j}(1+\delta_j) + O(N\epsilon^2)\sum|x_j|$, 가 되며, 이 때 $|\delta_j < 2\epsilon$ 이다.

만일 그냥 $\sum x_j$ 으로 계산하였을 때, 계산된 결과는 $\sum x_j(1+\delta_j)$ 가 되는데 (단 $|\delta_j| < (n-j)\epsilon$) 이를 Kahan 합 공식과 비교해 본다면 정말 엄청난 차이가 아닐 수 없습니다. 각각의 합에서 오직 2ϵ 밖에 오차가 발생하지 않지만, 그냥 단순한 덧셈으로 할 경우 각 합에서 최대 $n\epsilon$ 까지나 차이가 발생하게 때문입니다. 위 정리의 자세한 내용은 덧셈에서의 오차를 참고하시기 바랍니다.

만일 최적화기가 부동 소수점 연산에서도 대수의 법칙을 적용하여 $C = [T-S]-Y = [(S+Y)-S]-Y = 0$ 으로 만들어 알고리즘을 무용지물로 만들어버릴 수 있습니다. 여태까지 다룬 예제들은 최적화기를 만들 때, 비록 수학적으로 실수들 사이에서 성립하는 대수학 법칙을 적용한다 하더라도, 부동 소수점 변수들에 적용할 때에는 매우 주의를 기울여야 한다는 뜻입니다.

최적화기가 부동 소수점 코드의 의미를 바꿀 수 있는 도 다른 경우로 코드에 상수들이 있을 때 입니다. 식 `1.0E-40*x` 에서, 암시적으로 십진수에서 이진수 상수로의 변환이 이루어집니다. 이 때, 이 상수는 이진수로 정확히 변환이 불가능 하기 때문에 부정확함 예외가 발생하게 됩니다. 게다가 만일 이 식이 단정밀도로 계산되었다면, 언더플로우 플래그 역시 설정이 됩니다. 따라서 이 상수가 부정확하기 때문에, 이 이진수의 정확한 변환은 IEEE 반올림 모드에 의해 어떻게 변환이 될 지 달려있습니다. 따라서 만일 최적화기가 1.0E-40 을 컴파일 타임에 변환하게 된다면, 이는 프로그램의 의미를 변경하게 되는 것이 됩니다. 하지만, 27.5 와 같은 수들은 안전하게 컴파일 타임에 변환될 수 있고 항상 정확하기 때문에 어떠한 예외도 발생시키지 않고, 반올림 모드에도 영향을 받지 않습니다. 만일, 컴파일 타임에 변환을 하고 싶은 상수들이 있다면, `const pi = 3.14159265` 처럼 상수 선언을 사용하도록 합니다.

부동 소수점 코드 의미를 바꿀 수 있는 최적화로 공통 부분식 제거 (Common subexpression elimination) 라는 것이 있습니다. 아래 코드를 한 번 보시면

```
C = A * B;
RndMode = Up
```

```
D = A * B;
```

비록 `A*B` 가 공통된 부분식으로 보일지어도 중간에 반올림 방식이 바뀌었기 때문에 두 개의 식은 다른 결과를 내게 됩니다. 그 외에도, $x = x$ 는 `true` 로 바뀔 수 없다던지 (왜냐하면 x 가 NaN 이면 성립하지 않으니까), `-x = 0 - x` 가 $x = +0$ 일 때 성립하지 않는다던지, $x < y$ 는 $x \geq y$ 의 역이 아니라던지 (NaN 의 경우) 등을 기억하시면 됩니다.

이러한 예들에도 불구하고, 부동 소수점 코드에 적용할 수 있는 유용한 최적화 기법들은 물론 있습니다. 첫 번째로, 모든 부동 소수점 수들에게 성립하는 대수 항등식들이 있습니다. IEEE 산술에서의 예시로 $x+y = y+x, 2\times x = x+x, 1\times x = x, 0.5\times x = x/2$ 등은 성립합니다.

마지막 예시로 `dx = x*y` 문장을 살펴봅시다. 여기서 x, y 는 모두 단정밀도 변수들이고, dx 는 배정밀도 변수 입니다. 두 개의 단정밀도를 입력 받아서 그 곱셈 결과를 배정밀도로 리턴하는 명령이 명령어 세트에 있다면, 굳이 피연산자들을 배정밀도로 바꾼 다음 배정밀도 곱셈을 수행하는 단계를 거치지 않고도 그대로 매핑될 수 있습니다.

일부 컴파일러 제작자들은 $(x+y)+z$ 를 $x+(y+z)$ 로 변환하는 것을 제한하는 것을 불합리 하다고 생각합니다. 왜냐하면 아마도 그들은 부동 소수점 수들은 실수를 모델로 하였기 때문에, 부동 소수점 역시 실수들이 따르는 법칙들을 그대로 지켜야 한다고 생각하기 때문입니다. 하지만 실수의 특성을 그대로 구현해내는 것은 매우 어렵습니다. 만일 어떤 시스템에서 실수의 특성을 그대로 구현했다고 해봅시다. 그렇다면 만일 두 개의 n 비트 수들이 곱해진다면, 그 결과는 $2n$ 비트가 될 것입니다. 또한 두 n 비트 수들의 덧셈에서, 그 결과는 $n+$ 두 수의 지수 차이 만큼의 비트를 필요로 할 것입니다. 따라서 두 n 비트 수의 덧셈에서 그 결과는 최대 $(e^{max} - e^{min}) + n$ 비트, 대략 $2\times e^{max} + n$ 비트가 필요하게 됩니다. 따라서 수천개의 연산을 필요로 하는 알고리즘의 경우, 수들의 가수 부분의 크기가 점점 커지게 되며 결국 끔찍할 정도로 느려진 연산 속도를 볼 수 있을 것입니다. 게다가 $\sin, \cos$ 과 같은 함수들을 구현하기에는 더욱 어려울 것인데 왜냐하면 이 초월함수들의 값은 유리수가 아니기 때문입니다. 정확한 정수 연산은 lisp 시스템과 같은 여러 시스템에서 제공되며 몇몇 문제들을 다루는데 매우 편리하지만, 정확한 부동 소수점 연산은 거의 쓸모가 없습니다.

한 가지 중요한 사실은 여러 유용한 알고리즘들 (앞에서 나온 Kahan 의 합 공식) 은 $(x + y) + z \neq x + (y + z)$ 라는 사실을 사용하며, 다음과 같은 한계가 주어질 때 작동합니다.

$$a \oplus b = (a + b)(1 + \delta)$$

이러한 한계는 거의 대부분의 상업 하드웨어에서 성립하므로, 수치 프로그래머들이 이러한 알고리즘을 무시하는 것은 매우 멍청한 짓일 뿐더러, 컴파일러 제작자들이 부동 소수점 변수들을 마치 실수 처럼 취급하여 처리하여 이러한 알고리즘을 무용지물로 만드는 것은 정말 무책임한 행동입니다.

3.3 처리 (Handling)

여태까지 이야기 해왔던 것은 주로 정확도와 정밀도에 관한 이야기였습니다. 그 외에도 오류 처리기 또한 흥미로운 시스템 관련 주제 입니다. IEEE 표준에서는 5 종류의 예외들에 대해 사용자가 각각에 예외처리기 (trap handler) 를 할당할 수 있도록 강력히 권장하고 있습니다. 올바르지 못한 연산과, 0 으로 나눔 예외에서 예외처리기에는 피연산자들이나 정확히 반올림 된 결과값이 인자로 전달됩니다. 어떤 프로그래밍 언어를 사용하느냐에 따라, 예외 처리기는 프로그램의 다른 변수들에 접근할 수 도 있습니다. 모든 예외들에 대해 예외 처리기는 어디에서 어떤 연산이 실행되어서 예외가 발생하였는지에 대한 정보를 알아야만 합니다.

IEEE 표준은 모든 연산들이 이론상 연속적 (serial) 으로 처리된다고 가정하기 때문에, 인터럽트가 발생시에 어떤 작업이고 어떤 피연산자 인지 구분할 수 있도록 하였습니다. 하지만, 파이프라이닝 (pipelining) 이나 여러개의 산술 유닛을 가지고 있는 컴퓨터들의 경우 단순히 예외 처리기를 설치하는 것만으로 부족할 수 있습니다. 이러한 컴퓨터들의 경우에는 어떠한 연산에서 예외가 발생하였는지 정확하게 하드웨어 적으로 지원해줄 필요성이 있습니다.

파이프라이닝이란, 한 명령어 처리가 끝나기 전에 다른 명령어의 수행을 시작하는 방법으로, 높은 성능을 낼 수 있습니다.

다른 예시로 아래와 같은 프로그램 코드의 일부분을 살펴봅시다.

```
x = y * z;
z = x * w;
a = b + c;
d = a / x;
```

만일 두 번재 곱셈에서 예외가 발생하였고 예외 처리기가 `a` 의 값을 필요로 한다고 생각해봅시다. 만일 덧셈과 곱셈을 동시에 할 수 있는 하드웨어에서는 아마 최적화기는 덧셈 연산을 두 번째 곱셈 연산 위로 재배치 하여 덧셈과 첫번째 곱셈이 동시에 연산될 수 있도록 조정하였을 것입니다. 따라서 두 번째 곱셈에서 예외가 발생하였을 때 이미 `a = b + c` 가 이미 실행된 이후이기 때문에 `a` 의 값을 이미 바꿔버렸을 것입니다. 사실 컴파일러 입장에서는 이러한 형태의 최적화를 하지 않도록 하는 것도 말이 안되는 것이, 모든 부동 소수점 연산은 예외를 발생할 가능성이 있기 때문에 사실상 모든 연산 스케쥴링 최적화가 불가능 하게 된다는 말과 동일하기 때문입니다. 이 문제는 예외 처리기가 프로그램의 임의의 변수에 직접적으로 접근하는 것을 막는 것으로 피해갈 수 있습니다. 그 대신에 예외 처리기에는 인자로 피연산자들과 그 연산 결과값 만을 가질 수 있도록 하면 됩니다.

그럼에도 불구하고 여전히 문제가 있는데, 아래와 같은 코드를 보면

```
x = y*z;
z = a + b;
```

위 역시 2 개의 연산들이 동시에 수행될 수 있습니다. 그런데 만약 곱셈에서 예외가 발생하게 된다면, 인자 `z` 는 이미 덧셈으로 인해 값이 덮어 씌어질 것입니다. (특히,

덧셈이 곱셈보다 더 빠르다) 따라서 IEEE 표준을 지원하는 컴퓨터 시스템에서는 반드시 z 의 값을 저장해 놓는 방법을 마련해놓던지, 아니면 하드웨어 혹은 컴파일러로 하여금 이러한 상황이 발생하는 것을 아예 막아 놓는 수 밖에 없습니다.

W.Kahan 은 선치환(presubstitution) 이라는 기법을 사용해서 예외 처리기들이 위와 같은 문제에 처하지 않을 수 있게 할 수 있다고 제안하였습니다. 이 방법은 사용자가 예외가 발생하였을 때, 사용하고 싶은 값과 예외를 지정할 수 잇도록 하였습니다. 예를 들어서 $(\sin x)/x$ 를 계산하는 코드가 있다고 하고, 사용자가 $x = 0$ 을 넣는 경우는 매우 드물기 때문에, x 가 0 인지 검사하는 테스트를 생략하고, $0/0$ 이 발생하는 경우만 따로 처리하자고 생각하였습니다. 따라서 사용자는 1 을 리턴하는 IEEE 예외 처리기를 제작해서 설치하여 $\sin x/x$ 를 계산하기 전에 예외 발생 시에 1 을 리턴하도록 하였을 것입니다. 한편 선치환 기법을 사용한다면, 사용자는 올바르지 못한 연산이 발생 시에 1 을 사용하도록 지정할 수 있습니다. Kahan 이 이를 '선치환' 이라 이름붙인 이유는 이 리턴되는 값은 반드시 예외가 발생하기 전에 지정되어야 하기 때문입니다. 반면에 예외 처리기를 이용하는 경우에는 예외가 발생하였을 때 리턴될 값이 계산될 수 있습니다.

선치환의 장점은 하드웨어 적으로 구현하기에 쉽다는 점에 있습니다. 어떠한 예외가 발생하였는지 결정되었다면, 그 예외를 인덱스로 이용하여 원하는 연산의 결과를 포함하고 있는 테이블에서 그 결과값을 찾을 수 있기 때문이빈다. 비록 선치환 기능이 여러 매력적인 부분이 있지만, IEEE 표준이 널리 사용되어 있기 때문에 많은 하드웨어 제작사들에서 이 기능을 구현하기에는 어려울 것으로 보입니다.

제 4 장

자세한 증명들

이 문서에서 부동 소수점 연산의 성질들에 관해 몇 가지 이야기 했었습니다. 우리는 이제부터 부동 소수점이 수학적으로 엄밀하게 증명할 수 있다는 것을 보일것입니다. 이 장은 3 개의 섹션으로 구분이 되는데, 첫 번째 반올림 오차 에서는 오차 분석에 대한 기본적인 소개와 에서 다루었던 정리들에 관한 증명을 할 것입니다. 두 번째 이진수를 십진수로 변환하기 에서는 이진수를 십진수로 변환하는 과정에 대해 알아볼 것이며, 에서 몇 가지 넘어 갔던 부분을 채울 것입니다. 그리고 마지막 덧셈에서의 오차 에서는 Kahan 의 합 공식에 대해 다룰 것입니다.

4.1 반올림 오차

이전에 반올림 오차에 대해서 이야기 했었을 때, 덧셈과 뺄셈에서 보호 숫자 한 자리 만으로 그 결과가 정확하다고 보장할 수 있었습니다. (정리 1.3.2) 우리는 이제 이 정리를 증명할 것입니다. 정리 1.3.2 는 2 개의 부분으로 구성되어 있습니다. 하나는 덧셈에 관한 것이고, 하나는 뺄셈에 관한 것입니다. 일단 먼저 뺄셈에 대한 부분을 보자면

Theorem 4.1.1. 만일 x, y 가 β, p 를 인자로 가지는 부동 소수점 수 이고, 뺄셈이 $p+1$ 자리수 *(즉 1 개의 보호 숫자)* 로 이루어질 때, 뺄셈 결과의 상대 오차는 $(\frac{\beta}{2}+1)\beta^{-p} = (1+\frac{2}{\beta})e \leq 2e$ 보다 작다.

Proof. 일단 $x > y$ 가 되도록 x, y 를 설정하도록 하자. 또한, x, y 의 크기를 조정해서, x 가 $x_0.x_1...x_{p-1} \times \beta^0$ 꼴이 되도록 하자. 만일 y 가 $0.y_1y_2...y_p$ 꼴이라면, 보호 숫자 때문에 차이를 계산하였을 때 그 결과가 정확히 차이를 계산한 것을 반올림 한 것과 동일하다고 보장이 되므로 이 경우 반올림 오차가 최대 e 일 것입니다. 일반적인 경우 $y = 0.0...0y_{k+1}..y_{k+p}$ 라 두고, $\bar{y}$ 는4 y 를 맨 앞 $p+1$ 자리만 잘라낸 것이라고 하자. 그렇다면

$\bar{y} = 0.0..0y_{k+1}..y_{p+1}$

$$y - \bar{y} < (\beta - 1)(\beta^{-p-1} + \beta^{-p-2} + \cdots + \beta^{-p-k}) \tag{4.1}$$

가 성립합니다. 이 때, 보호 숫자의 정의에 따라 $x-y$ 의 계산된 값은 $x-\bar{y}$ 를 가장 가까운 부동 소수점 수인 $(x-\bar{y})+\delta$ 로 반올림 한 것과 같고, 이 때 반올림 오차 δ 는

$$|\delta| \leq (\beta/2)\beta^{-p} \tag{4.2}$$

을 만족합니다. 정확한 차이는 $x-y$ 이므로, 오차는 $(x-y)-(x-\bar{y}+\delta) = \bar{y}-y+\delta$ 가 됩니다. 이는 세 가지 경우로 나눌 수 있는데, 만일 $x-y \geq 1$ 이라면, 상대 오차는

$$\frac{y-\bar{y}+\delta}{1} \leq \beta^{-p}[(\beta-1)(\beta^{-1}+\cdots+\beta^{-k})+\beta/2] < \beta^{-p}(1+\beta/2) \tag{4.3}$$

두번재 경우로 만일 $x-\bar{y}<1$ 이라면 $\delta=0$ 입니다. 이 때 $x-y$ 가 가장 작아질 수 잇는 경우는

$$1.0-0.(\overbrace{0\cdots0}^{k})(\overbrace{\rho\cdots\rho}^{p}) > (\beta-1)(\beta^{-1}+\cdots+\beta^{-k}), \text{where } \rho=\beta-1 \tag{4.4}$$

이 경우 상대 오차의 범위는

$$\frac{y-\bar{y}+\delta}{(\beta-1)(\beta^{-1}+\cdots+\beta^{-k}} < \frac{(\beta-1)\beta^{-p}(\beta^{-1}+\cdots+\beta^{-k})}{(\beta-1)(\beta^{-1}+\cdots+\beta^{-k})} = \beta^{-p} \tag{4.5}$$

마지막 경우로는 $x-y<1$ 이지만 $x-\bar{y}\geq 1$ 일 때 인데, 이 것이 가능할 유일한 경우는 $x-\bar{y}=1$ 일 때 밖에 없고 이 경우 $\delta=0$ 입니다. 그런데 $\delta-0$ 이면 식 4.5 을 다시 적용하여, 결국 다시 상대 오차는 $\beta^{-p}<\beta^{-p}](1+\beta/2)$ 로 제한되게 됩니다. □

왜 $\delta=0$ 이 되냐면, $x-\bar{y}$ 의 맨 앞자리가 0 이 되므로, 지수 값을 바꿔서 가수 부분을 1 비트 더 표현 가능하게 됩니다. 따라서 반올림이 필요 없어지므로 $\delta=0$ 이 됩니다.

$\beta=2$ 인 경우, 그 한계는 정확히 $2e$ 이며, 등호조건은 $x=1+2^{2-p}, y=2^{1-p}-2^{1-2p}$ 일 때 $p\to\infty$ 입니다. 또한 아래 결과에 따르면 같은 부호의 숫자들을 더할 때면 높은 정확도를 위해 굳이 부호 숫자를 살용할 필요가 없습니다.

(*) 캐리 아웃 이란, 연산을 수행하는데, 연산 밖으로 받아올림이 나가는 것을 의미합니다. 예를 들어 3 비트 자료형에서 $100+100$ 을 하게 되면 000 이 되고, 1 이 캐리 아웃 됩니다

Theorem 4.1.2. 만일 $x\geq 0, y\geq 0$ 이라면, 보호 숫자를 사용하지 않고 $x+y$ 를 계산해도 가능한 최대의 상대오차는 2ϵ 이다.

Proof. 일반성을 잃지 않고 $x\geq y\geq 0$ 이라고 해봅시다. 그리고, x 의 크기를 $d.dd...d\times\beta^0$ 이 되도록 조정해도 무방합니다. 먼저 y 가 캐리 아웃(carry out)이 발생되지 않았다(*)고 가정합시다. 그렇다면 y 의 쉬프트 된 끝 부분의 값은 β^{-p+1} 보다 작을 것이고, (**) 둘의 덧셈의 크기는 최소 1 일 것이므로, 상대 오차는 $\beta^{-p+1}/1 = 2e$ 보다 작을 것입니다. 만일 캐리 아웃이 발생하였다면, 쉬프트로 인한 오차가 반올림으로 인한 오차 $\frac{1}{2}\beta^{-p+2}$ 가 반올림 오차에 더해져야 합니다. 이 때, 그 합은 최소 β 이므로, 상대 오차는

$$(\beta^{-p+1}+\frac{1}{2}\beta^{-p+2})/\beta = (1+\beta/2)\beta^{-p} \leq 2\epsilon$$

이 되어 성립합니다. □

(**) 쉬프트 된 끝이 아무리 커봐야 $y_0.y_1...y_{p-1}\times\beta^0$ 이므로, 그 끝 부분의 합은 최대 $\beta^{-p}+\beta^{-p-1}+\ldots < \beta^{p-1}\frac{1}{1-1/\beta} = \frac{1}{\beta-1}\beta^{-p+1} \leq \beta^{-p+1}$ 이다.

위 두 정리를 종합하면 정리 1.3.2 가 나온다는 사실은 명백합니다. 정리 1.3.2 는 한 개의 연산을 할 때 발생하는 상대 오차에 대해 다룹니다. 반면에 x^2-y^2 와 $(x+y)(x-y)$ 를 수행할 대의 상대오차를 비교하기 위해서는 여러 기본 연산을 같이 수행할 때의 상대 오차를 계산해야겠지요. $x \ominus y$ 의 상대 오차는 $\delta_1 = [(x \ominus y) - (x-y)]/(x-y)$ 이고 이는 $|\delta_1| \geq 2e$ 를 만족합니다. 다르게 쓰자면

$$x \ominus y = (x-y)(1+\delta_1), |\delta_1| \geq 2\epsilon \tag{4.6}$$

비슷하게,

$$x \oplus y = (x+y)(1+\delta_2), |\delta_2| \geq 2\epsilon \tag{4.7}$$

를 알 수 있습니다. 곱셈의 경우, 정확히 계산을 한 뒤에 반올림이 된다고 가정하면, 그 상대 오차는 최대 .5ulp 일 것이므로, 모든 부동 소수점 수 u, v 에 대해서

$$u \otimes v = uv(1+\delta_3), |\delta_3| \geq \epsilon \tag{4.8}$$

가 성립하게 됩니다. 이 식들을 모두 모아서 $(u = x \ominus y, v = x \oplus y)$ 합치면,

$$(x \ominus y) \otimes (x \oplus y) = (x-y)(1+\delta_1)(x+y)(1+\delta_2)(1+\delta_3) \tag{4.9}$$

가 됩니다. 따라서, $(x-y)(x+y)$ 를 계산할 때 발생하는 상대 오차는

$$\frac{(x \ominus y) \otimes (x \oplus y) - (x^2-y^2)}{(x^2-y^2} = (1+\delta_1)(1+\delta_2)(1+\delta_3) - 1 \tag{4.10}$$

이므로 상대오차는 $\delta_1 + \delta_2 + \delta_3 + \delta_1\delta_2 + \delta_1\delta_3 + \delta_2\delta_3 + \delta_1\delta_2\delta_3$ 이 되며, 이의 한계는 $5\epsilon + 8\epsilon^2$ 입니다. 다시 말해, 최대 상대 오차는 대략 5 반올림 오차 정도 됩니다. (참고로 ϵ 는 작은 수 이기 때문에 ϵ^2 는 무시할 수 있습니다)

반면에 비슷한 방법으로 $(x \otimes x) \ominus (y \otimes y)$ 를 분석하면 상대오차로 작게 나올 수 없는데, 그 이유는 두 개의 비슷한 값인 x, y 가 $x^2 - y^2$ 에 넣었을 때 상대 오차는 꽤 크기 때문입니다. 위에서 한 방법과 동일하게 $(x \otimes x) \ominus (y \otimes y)$ 를 계산해보면

$$(x \otimes x) \ominus (y \otimes y) = [x^2(1+\delta_1) - y^2(1+\delta_2)](1+\delta_3) = ((x^2-y^2)(1+\delta_1) + (\delta_1-\delta_2)y^2)(1+\delta_3)$$

만일 x, y 값이 가까우면, 항 $(\delta_1 - \delta_2)y^2$ 은 결과값 $x^2 - y^2$ 만큼 커질 수 있습니다. 이러한 결과는 앞서 우리가 주장한 $(x-y)(x+y)$ 가 $x^2 - y^2$ 보다 더 정확하다 라는 것을 증명해줍니다.

이번에는 이전에 삼각형의 넓이를 구하는 공식을 분석해보도록 합시다. 식 1.6 을 계산하면서 발생하는 최대 오차를 추정하기 위해 다음과 같은 정리가 필요합니다.

Theorem 4.1.3. 만일 보호 숫자를 이용해서 뺄셈이 수행되고, $y/2 \leq x \leq 2y$ 라면, $x - y$ 는 정확하게 계산된다.

Proof. 만일 x, y 의 지수가 같다면 $x \ominus y$ 를 정확하게 계산할 수 있습니다. x, y 의 지수가 다르다면, 문제 조건에 따라 최대 1 만큼 차이가 날 수 있습니다. 일반성을 잃지 않고 $0 \leq y \leq x$ 라고 가정하면, (아니라면 크기를 조정하면 된다), x 는 $x_0.x_1...x_{p-1}$ 이고, y 는 $0.y_1...y_p$ 로 표현됩니다. 이제 $x \ominus y$ 를 계산하는 알고리즘은, $x - y$ 를 정확히 계산한 후 부동 소수점으로 반올림 하게 됩니다. 만일 그 차이가 $0.d_1...d_p$ 꼴이라면, 이미 그 차이는 p 자리 이므로 반올림 할 필요가 없겠지요. 그런데 $x \leq 2y, x - y \leq y$ 이고 y 는 $0.d_1d_2..d_p$ 꼴 이므로 성립하게 됩니다. □

그런데, $x \leq 2y, x - y \leq y$ 이 $\beta > 2$ 일 때, 정리 4.1.3 에서의 가정이 $y/\beta \leq x \leq \beta y$ 로 바뀔 수 없고, 더 엄격한 가정인 $y/2 \leq x \leq 2y$ 가 필요합니다. $(x - y)(x + y)$ 의 오차 분석에서 우리는 덧셈과 뺄셈 연산에서 상대 오차가 작다는 사실(정리 4.1.2)을 사용하였습니다. 이는 가장 간단한 형태의 식의 오차 분석인데요, 반면에 식 1.6 과 같이 좀 더 복잡한 식의 오차 분석은 정리 4.1.3 을 필요로 합니다.

Theorem 4.1.4. 만일 뺄셈이 보호 숫자를 이용해 수행되고, a, b, c 가 삼각형의 세 변의 길이 일 때 *($a \geq b \geq c$)*, $(a + (b + c))(c - (a - b))(c + (a - b))(a + (b - c))$ 의 식을 계산 할 때 발생하는 상대 오차의 크기는 최대 16ϵ 이다 *(*단, $\epsilon < .005$ *)*

Proof. 각각의 항을 하나씩 살펴봅시다. 정리 4.1.2 에 따르면 δ_1 가 상대 오차 일 대 $b \oplus c = (b + c)(1 + \delta_1)$ 이고 $|\delta_1| \leq 2\epsilon$ 이였습니다. 따라서, 첫 번째 항 $(a \oplus (b \oplus c)) = (a + (b \oplus c))(1 + \delta_2) = (a + (b + c)(1 + \delta_1))(1 + \delta_2)$ 이므로,

$$\begin{aligned}(a + b + c)(1 - 2\epsilon)^2 &\leq [a + (b + c)(1 - 2\epsilon)](1 - 2\epsilon) \\ &\leq a \oplus (b \oplus c) \\ &\leq [a + (b + c)(1 + 2\epsilon)](1 + 2\epsilon) \\ &\leq (a + b + c)(1 + 2\epsilon)^2\end{aligned}$$

임을 알 수 있습니다. 이 말은, 아래를 만족하는 η_1 가 존재한다는 의미 입니다.

$$(a \oplus (b \oplus c)) = (a + b + c)(1 + \eta_1)^2, |\eta_1| \leq 2\epsilon \tag{4.11}$$

그 다음 항은 c 와 $a \ominus b$ 사이의 나쁜 지워짐의 가능성 내포하고 있는 항인데, 왜냐하면 $a \ominus b$ 가 반올림 오차를 가지고 있을 수 있기 때문입니다. 그런데, a, b, c 모두 삼각형의 세 변이기 때문에 $a \leq b + c$ 이고, $c \leq b \leq a$ 라는 사실을 통해 $a \leq b + c \leq 2b \leq 2a$ 임을 알 수 있습니다. 따라서 $a - b$ 는 정리 4.1.3 을 만족하므로, $a - b = a \ominus b$ 가 정확하다는 의미가 됩니다. 따라서 $c \ominus (a - b)$ 는 아무런 문제가 없는 뺄셈이고, 따라서 정리 4.1.1 에 의해

$$(c \ominus (a \ominus b)) = (c - (a - b))(1 + \eta_2), |\eta_2| \leq 2\epsilon \tag{4.12}$$

와 같이 근사될 수 있습니다. 세번째 항의 경우 두 개의 정확한 양 수의 합이므로,

$$(c \oplus (a \ominus b)) = (c + (a - b))(1 + \eta_3), |\eta_3| \leq 2\epsilon \tag{4.13}$$

가 됩니다. 마지막으로 마지막 항의 경우 정리 4.1.1 와 4.1.2 을 사용해서 아래와 같이 표현할 수 있게 됩니다.

$$(a \oplus (b \ominus c)) = (a + (b - c))(1 + \eta_4)^2, |\eta_4| \leq 2\epsilon \tag{4.14}$$

만일, 곱셈이 정확히 반올림 된다고 가정하면, $x \otimes y = xy(1+\zeta), |\zeta| \leq \epsilon$ 가 되므로, 식 4.11, 4.12, 4.13, 4.14 를 조합하면

$$\begin{aligned}&(a \oplus (b \oplus c))(c \ominus (a \ominus b))(c \oplus (a \ominus b))(a \oplus (b \ominus c)) \\ &\leq (a + (b + c))(c - (a - b))(c + (a - b))(a + (b - c))E\end{aligned}$$

이고, 여기서 E 는

$$E = (1 + \eta_1)^2(1 + \eta_2)(1 + \eta_3)(1 + \eta_4)^2(1 + \zeta_1)(1 + \zeta_2)(1 + \zeta_3)$$

가 됩니다. E 상한선은 $(1+2\epsilon)^6(1+\epsilon)^3$ 이 되는데 식을 전개하면 $1+15\epsilon+O(\epsilon^2)$ 가 됩니다. 어떤 사람들은 단순히 $O(e^2)$ 항을 무시하곤 하는데, 사실 이를 고려해도 크게 여려울 것은 없습니다. $(1+2\epsilon)^6(1+\epsilon)^3) = 1+15\epsilon+\epsilon R(\epsilon)$ 이라 하면, $R(\epsilon)$ 은 인자로 e 에 대한 양계수 다항식이 됩니다. 따라서 ϵ 에 대한 증가 함수가 되지요. 그런데 모든 $\epsilon < .005$ 인 ϵ 에 대해서, $R(.005) = .505, R(\epsilon) < 1$ 가 되므로 $E \leq (1 + 2\epsilon)^6(1 + \epsilon)^3 < 1 + 16\epsilon$ 가 됩니다. E 의 하한선을 구하기 위해서는 $1 - 15\epsilon - \epsilon R(\epsilon) < E$ 임에서 착안해서, $\epsilon < .005$ 일 때 $1 - 16\epsilon < (1 - 2\epsilon)^6(1 - \epsilon)^3$ 가 됩니다. 이 식들을 종합하면 E 의 범위인 $1 - 16\epsilon < E < 1 + 16\epsilon$ 를 구할 수 있게 됩니다. 따라서 상대 오차의 최대값은 16ϵ 가 됩니다. □

정리 4.1.4 는 식 1.6 에서 명백하게 나쁜 지워짐이 발생하지 않는다는 것을 보여주고 있습니다. 따라서, 수치적으로 식 1.6 이 안정하다는 사실을 보일 필요 없이, 전체 식이 어떠한 범위 안에 구속되어 있다는 사실을 말할 수 있게 됩니다. 이 사실은 정리 1.4.1 에서 보여주는 바와 같지요. 아래는 정리 1.4.1 의 증명 입니다.

Proof.

$$q = (a + (b + c))(c - (a - b))(c + (a - b))(a + (b - c))$$

라 하고,

$$Q = (a \oplus (b \oplus c)) \otimes (c \ominus (a \ominus b)) \otimes (c \oplus (a \ominus b)) \otimes (a \oplus (b \ominus c))$$

라 정의합시다. 그리고 정리 4.1.4 에서 $\delta \leq 16\epsilon$ 일 때 $Q = q(1+\delta)$ 임을 보였습니다. 또한

$$1 - 0.52|\delta| \le \sqrt{1-|\delta|} \le \sqrt{1+|\delta|} \le 1 + 0.52|\delta| \tag{4.15}$$

이므로 $\delta \le .04/(.52)^2 \approx .15$ 이고 $|\delta| \le 16\epsilon \le 16(.005) = .08$ 이므로 δ 가 조건을 만족함을 알 수 있습니다. 따라서 $|\delta_1| \le .52|\delta| \le 8.5\epsilon$ 일 때 $\sqrt{Q} = \sqrt{q(1+\delta)} = \sqrt{q}(1+\delta_1)$ 임을 알 수 있습니다. 만일 제곱근이 .5ulp 이내로 계산된다면, $\sqrt{Q}$ 를 계산할 때 발생하는 오차는 $\delta_2| \le \epsilon$ 일 때 $(1+\delta_1)(1+\delta_2)$ 가 됩니다. 만일 $\beta = 2$ 라면, 4 로 나눌 때 발생하는 오차가 없겠지요. 아니라면, 나눗셈을 위해 한 개의 항 $|\delta_3| \le \epsilon$ 을 만족하는 $1+\delta_3$ 이 더 필요하고, 정리 4.1.4 를 증명할 때 사용한 기법을 이용해 최종 오차$(1+\delta_1)(1+\delta_2)(1+\delta_3)$ 의 지배적인 항이 $1+\delta_4$ 임을 알 수 있습니다. (단, $|\delta_4| \le 11\epsilon$) □

이번에는 정리 1.4.2 에서 간단하게만 설명하고 생략한 증명을 아래 정리에서 좀 더 정확하게 짚고 넘어가도록 합시다.

Theorem 4.1.5. 만일 $\mu(x) = \ln(1+x)/x$ 일 때, $0 \le x \le \frac{3}{4}$, $\frac{1}{2} \le \mu(x) \le 1$ 이라면, μ 의 미분 $\mu'(x)$ 는 $|\mu'(x)| \le \frac{1}{2}$ 를 만족합니다.

교대 급수 (alternating series) 란, 부호가 계속 바뀌는 급수를 의미합니다

Proof. $\mu(x)$ 의 전개식이 감소 교대급수 로 주어지기 때문에 $(\mu(x) = 1 - x/2 + x^2/3 - ...)$, $x \le 1, \mu(x) \ge 1 - x/2 \ge 1/2$ 임을 알 수 있습니다. 또한, μ 의 급수가 교대하기 때문에 $\mu(x) \le 1$ 임을 쉽게 알 수 있습니다. 또한 $\mu'(x)$ 의 테일러 급수 또한 진동하며, $x \le \frac{3}{4}$ 일 때 항이 감소하므로 $-\frac{1}{2} \le \mu'(x) \le -\frac{1}{2} + 2x/3$ 혹은 $-\frac{1}{2} \le \mu'(x) \le 0$ 임을 알 수 있습니다. 따라서 $|\mu'(x)| \le \frac{1}{2}$ 가 됩니다. □

이를 토대로 정리 1.4.2 의 증명을 해보도록 합시다.

Proof. ln 의 테일러 급수가

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - ...$$

로 교대급수 이기 때문에 $0 < x - \ln(1+x) < x^2/2$ 가 되므로 $\ln(1+x)$ 를 x 로 근사할 때 발생하는 상대 오차는 $x/2$ 로 제한됩니다. 만일 $1 \oplus x = 1$ 이라면 $|x| < \epsilon$ 이므로 상대 오차는 $\epsilon/2$ 로 제한됩니다.

만일 $1 \oplus x \neq 1$ 이라면, $\hat{x}$ 를 $1 \oplus x = 1 + \hat{x}$ 로 정의해봅시다. 그렇다면 $0 \le x < 1, (1 \oplus x) \ominus 1 = \hat{x}$ 입니다. 만일, 나눗셈과 로그 계산이 $\frac{1}{2}$ ulp 이내로 계산된다면, $\ln(1+x)/((1+x)-1)$ 의 계산된 결과값은

평균값 정리 (mean value theorem); 함수 $f(x)$ 가 닫힌 구간 $[a,b]$ 에서 연속이고 열린 구간 (a,b) 에서 미분 가능이면 $f'(c) = \frac{f(b)-f(a)}{b-a}$ 인 c 가 열린 구간 (a,b) 안에 존재한다

$$\frac{\ln(1 \oplus x)}{(1 \oplus x) \ominus 1}(1+\delta_1)(1+\delta_2) = \frac{\ln(1+\hat{x})}{\hat{x}}(1+\delta_1)(1+\delta_2) = \mu(\hat{x})(1+\delta_1)(1+\delta_2) \tag{4.16}$$

가 됩니다. 여기서 $|\delta_1| \le \epsilon$ 이고 $|\delta_2| \le \epsilon$ 이죠. $\mu(\hat{x})$ 의 값을 계산하기 위해, 평균값 정리를 사용하면,

$$\mu(\hat{x}) - \mu(x) = (\hat{x} - x)\mu'(\zeta) \tag{4.17}$$

를 만족하는 x 와 $\hat{x}$ 사이에 존재하는 ζ 를 찾을 수 있습니다. $\hat{x}$ 의 정의에 따라, $|\hat{x} - x| \leq \epsilon$ 이고, 정리 4.1.5 의 결과인 $|\mu(\hat{x} - \mu(x))| \leq \epsilon/2$ 혹은 $|\mu(\hat{x})/\mu(x) - 1| \leq \epsilon/(2|\mu(x)|) \leq \epsilon$ 을 종합하게 되면, $\mu(\hat{x}) = \mu(x)(1 + \delta_3)$ 임을 알 수 있습니다. (단 $|\delta_3| \leq \epsilon$). 마지막으로 x 를 곱하는 연산에서 마지막 δ_4 가 발생하게 되는데, 따라서 최종적으로 $x \ln(1 \oplus x)/((1 \oplus x) \ominus 1)$ 의 값은

$$\frac{x \ln(1+x)}{(1+x)-1}(1+\delta_1)(1+\delta_2)(1+\delta_3)(1+\delta_4), |\delta_i| \leq \epsilon$$

가 됩니다. 그리고 $\epsilon < 0.1$ 이면 $|\delta| \leq 5\epsilon$ 인 $(1+\delta_1)(1+\delta_2)(1+\delta_3)(1+\delta_4) = 1+\delta$ 가 됨을 쉽게 보일 수 있습니다. (앞 정리들에서 한 방법들과 동일) □

이차방정식의 근의 공식 $(-b \pm \sqrt{b^2 - 4ac})/2a$ 에서 식 4.6, 4.7, 4.8 에서 볼 수 있습니다. 이전에 지워짐 (Cancellation) 에서, $\pm$ 연산으로 인해 발생할 가능성이 생겼던 지워짐 문제를 수식을 다시 씀으로써 어떻게 해결하였는지 이야기하였습니다. 하지만, $d = b^2 - 4ac$ 를 계산 할 때 도 다른 지워짐 문제가 발생할 수 있는데요, 이는 단순히 식을 재배치 함으로써 해결할 수 없습니다. 간단히 말해서, 만일 $b^2 \approx 4ac$ 일 경우, 반올림 오차는 근의 공식으로 계산된 값 자리수의 절반 까지 신뢰할 수 없게 만들 수 있습니다. 아래는 이 사실을 간단히 증명한 것입니다.

Proof. $(b \otimes b) \ominus (4a \otimes c) = (b^2(1+\delta_1) - 4ac(1+\delta_2))(1+\delta_3)$ 이라 할 수 있고, 이 때, $|\delta_i| \leq \epsilon$ 입니다. 이 때 $d = b^2 - 4ac$ 로 치환, 이는 $(d(1+\delta_1) - 4ac(\delta_2 - \delta_1))(1+\delta_3)$ 으로 다시 쓸 수 있습니다. 오차의 근사치를 계산하기 위해, 이차이상의 δ_i 항들을 무시하면, 절대 오차는 $d(\delta_1 + \delta_3) - 4ac\delta_4$ 라고 쓸 수 있고, 이 때 $|\delta_4| = |\delta_1 - \delta 2| \leq 2\epsilon$ 이라 할 수 있습니다. $d << 4ac$ 이므로, 첫번째 항 $d(\delta_1 + \delta_3$ 은 무시할 수 있습니다. 따라서 두번째 항을 근사하기 위해 $ax^2 + bx + c = a(x - r_1)(x - r_2)$ 라 한다면, $ar_1r_2 = c$ 가 됩니다. 그런데, $b^2 \approx 4ac$ 이므로, $r_1 \approx r_2$ 가 되므로, 두 번째 항은 $4ac\delta_4 \approx 4a^2r_1^2\delta_4$ 가 됩니다. 따라서, $\sqrt{d}$ 의 계산된 값은 $\sqrt{d + 4a^2r_1^2\delta_4}$ 가 됩니다. 아래 부등식

$$p - q \leq \sqrt{p^2 - q^2} \leq \sqrt{p^2 + q^2} \leq p + q, p \geq q > 0$$

을 통해 $\sqrt{d + 4a^2r_1^2\delta_4} = \sqrt{d} + E$ 이고, $|E| \leq \sqrt{4a^2r_1^2|\delta_4|}$ 가 됩니다. 따라서 $\sqrt{d}/2a$ 의 절대 오차는 대략 $r_1\sqrt{\delta_4}$ 가 됩니다. 이 때 $\delta_4 \approx \beta^{-p}, \sqrt{\delta_4} \approx \beta^{-p/2}$ 이므로, $r_1\sqrt{\delta_4}$ 의 절대 오차는 근 $r_1 \approx r_2$ 의 절반 이하의 비트들을 신뢰할 수 없게 만듭니다. □

이번에는 정리 1.5.2 의 증명으로 돌아가봅시다. 이는 정리 4.1.6 와 3.2.1 을 기반으로 하고 있습니다.

Theorem 4.1.6. $0 < k < p$ 라 하고, $m = \beta^k + 1$ 이라고 하자. 이 때 모든 부동소수점 연산은 정확히 반올림 된다고 가정한다. 그러면, $(m \otimes x) \ominus (m \otimes x \ominus x)$ 는

정확히 x 가 $p-k$ 의 자리의 가수로 반올림 된 것과 같습니다. 여기서 $p-k$ 자리로 반올림 된다는 말은, x 의 가수 부분을 따로 떼서 하위 k 비트 왼쪽에 소수점을 붙이고 정수로 반올림 한 것과 같습니다.

즉 $x = d_1.d_2...d_n$ 이라면, x 를 반올림 해서 $\hat{x} = d_1.d_2...d_{p-k}$ 로 만든 것

이제 정리 1.5.2 을 증명 해보도록 합시다.

Proof. 정리 4.1.6 에 의해 x_h 는 x 가 $p-k = \lfloor p/2 \rfloor$ 자리로 반올림 됩니다. 만일, 캐리 아웃이 없었다면, 자명히도 x_h 는 $\lfloor p/2 \rfloor$ 자리의 가수로 표현할 수 있겠지요. 따라서, 캐리 아웃이 있었다고 가정합시다. $x = x_0.x_1...x_{p-1} \times \beta^e$ 라면, 반올림 시에 x_{p-k-1} 에 1 을 더하게 되고, 받아올림이 발생할 유일한 방법은 $x_{p-k-1} = \beta - 1$ 일 경우 밖에 없습니다. 그러면, x_h 의 맨 마지막 숫자는 $1 + x_{p-k-1} = 0$ 이 되므로 x_h 역시 $\lfloor p/2 \rfloor$ 자리로 표현할 수 있게 됩니다.

반올림을 통해 캐리 아웃이 발생하려면 결과적으로 맨 아래 비트에서 받아 올림이 생겨서 쭉 앞까지 진행되는 방법 밖에 없다.

한편 x_l 을 처리 하기 위해 x 의 크기를 조정해서 $\beta^{p-1} \leq x \leq \beta^p - 1$ 로 만들 수 있습니다. $x = \bar{x_h} + \bar{x_l}$ 이고, $\bar{x_h}$ 는 x 의 상위 $p-k$ 자리, $\bar{x_l}$ 을 하위 k 자리라 정의합시다. 그러면, 3 가지 경우만 고려하면 충분한데 먼저 $\bar{x_l} < (\beta/2)\beta^{k-1}$ 인 경우 x 를 $p-k$ 자리로 반올림 하는 것은 단순히 잘라내는 것이므로 $x_h = \bar{x_h}$ 이고, $x_l = \bar{x_l}$ 이라 할 수 있습니다. $\bar{x_l}$ 은 최대 k 자리까지 있기 때문에 만일 p 가 짝수라면 $\bar{x_l}$ 은 최대 $k = \lceil p/2 \rceil = \lfloor p/2 \rfloor$ 자리가 됩니다. 아니면, $\beta = 2$ 이고 $\bar{x_1} < 2^{k-1}$ 은 $k - 1 \leq \lfloor p/2 \rfloor$ 비트의 가수로 표현 가능하게 됩니다. 두 번째 경우는 $\bar{x} > (\beta/2)\beta^{k-1}$ 로, x_h 를 계산 할 때 반올림이 발생하므로 $x_h = \bar{x_h} + \beta^k$ 이고, $x_1 = x - x_h = x - \bar{x_h} - \beta^k = \bar{x_l} - \beta^k$ 가 됩니다. 그렇다면 $\bar{x_l}$ 은 최대 k 자리가 되어서, $\lfloor p/2 \rfloor$ 자리수로 표현 가능하게 됩니다. 마지막으로 $\bar{x_l} = (\beta/2)\beta^{k-1}$ 일 경우 인데, 이 경우 어디에서 반올림이 일어나냐에 따라 $x_h = \bar{x_h}$or $\bar{x_h} + \beta^k$ 가 됩니다. 따라서 x_l 은 $(\beta/2)\beta^{k-1}$ 혹은 $(\beta/2)\beta^{k-1} - \beta^k = -\beta^k/2$ 가 되어서 모두 1 자리로 표현 가능하게 됩니다. □

이 것과 비슷한 공식으로 합을 정확히 표현하는 공식이 있습니다. 만일 $|x| \geq |y|$ 이라면 $x+y = (x \oplus y)+(x \ominus (x \oplus y)) \oplus y$ 가 됩니다. 하지만, 정확히 반올림 되는 연산을 사용할 경우 이 공식은 $\beta = 2$ 일 때만 성립하고 $\beta = 10$ 일 때는 성립하지 않습니다. (예를 들어 $x = .99998, y = .99997$ 일 때)

4.2 이진수를 십진수로 변환하기

단정밀도가 $p = 24$ 이고 $2^{24} < 10^8$ 이므로, 여러분은 아마 단정밀도 이진수를 8 자리 십진수로 변환한다면 충분히 다시 원래의 이진수로 변환할 수 있을 것이라 생각할 것입니다. 하지만, 사실 이것은 틀립니다.

Theorem 4.2.1. 만일 *IEEE* 단정밀도 수가 가장 가까운 *8* 자리 십진수로 변환되었다면, 이 변환된 십진수로 부터 원래의 이진수로 항상 변환 가능한 것은 아니다. 하지만, *9* 자리 십진수를 사용하면, 이 십진수를 다시 가장 가까운 이진수로 변환시 원래의 부동 소수점 이진수를 구할 수 있게 된다.

Proof. 한쪽만 열린 구간 $[10^3, 2^{10}) = [1000, 1024)$ 에 들어 있는 이진 단정밀도 수들은, 소수점을 기준으로 왼쪽에는 10 비트, 오른쪽에는 14 비트가 있을 것입니다. 따라서, 이 구간 안에는 총 $(2^{10} - 10^3)2^{14} = 393,216$ 가지의 서로 다른 이진수들이 가능하게 됩니다. 만일 십진수가 8 자리로 표현된다면, $(2^{10} - 10^3)10^4 = 240,000$ 개의 십진수들이 이 구간안에 들어가게 됩니다. 그런데 393,216 개의 서로 다른 이진수들을 240,000 개의 십진수로 표현한다는 것은 불가능한 일입니다. 따라서 8 자리로는 단정밀도 이진수를 유일하게 대응시키기는 불가능합니다.

그렇다면 9 자리로는 충분하다는 것을 보이기 위해, 두 이진수의 차이가 항상 두 십진수 차이 보다 항상 크갇는 것을 보일 것입니다. 이는 각각의 십진수 N 에 대해, 구간 $[N - \frac{1}{2}ulp, N + \frac{1}{2}ulp]$ 에는 최대 1 개의 이진수만 들어가게 되죠. 따라서, 각각의 이진수는 유일한 십진수로 대응되고, 그 십진수 역시 유일한 이진수에 대응이 될 것입니다.

먼저 두 이진수 사이 간격이 십진수 사이 간격 보다 항상 크다는 것을 보이기 위해 구간 $[10^n, 10^{n+1}]$ 을 생각해봅시다. 이 구간에서는 두 개의 연속한 십진수 차이가 $10^{(n+1)-9}$ 입니다. m 이 $10^n < 2^m$ 을 만족하는 최소의 정수라 할 때, 구간 $[10^n, 2^m]$ 에서 이진수 사이의 간격은 2^{m-24} 가 되고 구간 안에서 점점 그 간격은 늘어날 것입니다. 따라서 $10^{(n+1)-9} < 2^{m-24}$ 임을 확인하면 충분합니다. 그런데, 사실 $10^n < 2^m$ 이므로 $10^{(n+1)-9} = 10^n 10^{-8} < 2^m 10^{-8} < 2^m 2^{-24}$ 가 되서 성립합니다. □

비슷한 논리로 배정밀도의 경우 십진수로 변환했을 때 원래의 이진수로 다시 변환할 수 있으려면 17 자리의 십진수가 필요한지 알 수 있습니다.

이진수 십진수 변환은 플래그 사용의 또 다른 훌륭한 예시가 됩니다. 이전에 정밀도 섹션을 상기해보자면, 원래의 십진수에서 원래의 이진수로 복구하기 위해 십진수 이진수 변환이 반드시 정확히 계산되어야만 합니다. 이 변환은 N 과 $10^{|P|}$ 을 확장 단정밀도로 곱한 후 단정밀도로 반올림함으로써 수행됩니다. 당연하게도 $N \cdot 10^{|P|}$ 는 정확히 계산할 수 없습니다. 정확히 계산해야 되는 것은 $round(N \cdot 10^{|P|})$ 로, 확장 단정밀도에서 단정밀도로 반올림 되어야 하는 부분입니다. 이것이 왜 정확히 수행될 수 없는지 보기 위해 $\beta = 10, p = 2$ 인 단정밀도와, $p = 3$ 인 확장 단정밀도를 생각해봅시다. 만일 곱이 12.51 이라면, 확장 단정밀도 곱셈 연산에서 12.5 로 반올림 될 것입니다. 그리고 단정밀도로 반올림 하게 되면 12 가 될 것입니다. 하지만 이 답은 정확하지 않는데, 만일 12.51 을 그대로 단정밀도로 반올림 하였다면 13 이 되었을 것이기 때문입니다. 이러한 오차는 두 번의 반올림으로 인해 발생하였습니다.

IEEE 플래그를 사용함으로써 이중 반올림은 다음과 같이 피할 수 있습니다. 먼저 현재 부정확함 플래그의 값을 저장한 후 리셋합니다. 그리고 반올림 모드를 0 으로의 반올림으로 설정합니다. 그리고 $N \cdot 10^{|P|}$ 를 수행합니다. 이제 현재의 부정확함 플래그의 새로운 값을 ixflag 에 저장하고, 원래의 반올림 모드 와 부정확함 플래그의 값으로 되돌립니다. 만일 ixflag 가 0 이라면 $N \cdot 10^{|P|}$ 는 정확하므로 $round(N \cdot 10^{|P|})$ 역시 마지막 비트 까지 정확할 것입니다. 만일 ixflag 가 1 이라면, 몇몇 0 으로의 반올림은 항상 잘라내기 때문에 몇몇 비트 들은 잘려 나갔을 것입니다. 따라서 곱셈 결과의 가수

부분은 $1.b_1...b_{22}b_{23}...b_{31}$ 처럼 생겼을 것입니다. 이중 반올림 오류는 $b_{23}...b_{31} = 10...0$ 일 때 발생하게 됩니다. 이 두 경우를 모두 해결하는 방법은 ixflag 와 b_{31} 을 논리적 OR 연산을 시켜 주는 것입니다. 그리고 $round(N \cdot 10^{|P|})$ 를 수행하게 되면 모든 경우에 대해 정확히 계산될 것입니다.

4.3 덧셈에서의 오차

최적화기 에서, 많은 수의 항의 덧셈을 정확히 계산하는데의 문제점을 이야기 하였습니다. 이를 해결하는 가장 단순한 방법으로, 정밀도를 두 배로 높여서 계산하는 것입니다. 정밀도를 두 배로 높임으로써 얼마나 덧셈의 정밀도가 높아지는지 대략적으로 알기 위해 $s_1 = x_1, s_2 = s_1 \oplus x_2..., s_i = s_{i-1} \oplus x_i$ 라고 정의해봅시다. 그러면 $|\delta_i| \leq \epsilon$ 이라 할 때, $s_i = (1 + \delta_i)(s_{i-1} + x_i)$ 가 되는데, δ_i 의 이차항 이상을 무시한다면

$$s_n = \sum_{j=1}^{n} x_j(1 + \sum_{k=j}^{n} \delta_k) = \sum_{j=1}^{n} x_j + \sum_{j=1}^{n} x_j(\sum_{k=j}^{n} \delta_k) \tag{4.18}$$

식 4.18 의 첫 번째 등호 관계에서 $\sum x_j$ 의 계산된 값은, x_j 의 근사치들을 정확히 덧셈한 값과 같다는 사실을 시사합니다. x_1 의 첫 번째 근사치는 실제 x_j 와 $n\epsilon$ 정도 차이가 나게 되지만, 맨 마지막 항인 x_n 의 경우 ϵ 밖에 차이가 안나게 됩니다. 식 4.18 의 두 번째 등호 관계에서, 그 오차 항은 $n\epsilon \sum |x_j|$ 으로 유계임을 알 수 있겠지요. 정밀도를 두 배로 늘리는 것은 ϵ 을 제곱하는 것과 같습니다. 따라서 만일 IEEE 배정밀도로 합이 계산된다면, $1/\epsilon \approx 10^{16}$ 이므로 보통 범위의 n 에 대해서는 $n\epsilon << 1$ 이 성립합니다. 따라서, 정밀도를 두 배로 늘리게 되는 것은 최대 근사 범위였던 $n\epsilon$ 을 $n\epsilon^2 << \epsilon$ 로 바꾸게 되므로 Kahan 합 공식(정리 3.2.1) 의 오차 한계 2ϵ 보다 더 훌륭한 방법이 되는 것입니다. (물론 단정밀도만 사용하면 Kahan 의 정리가 훨씬 훌륭합니다만)

Kahan 의 합 공식이 어떻게 작동하는지 이해하기 위해 그 과정을 나타내는 그림을 보시기 바랍니다.

매번 피연산자가 더해질 때, 다음 루프에 보정값 C 가 적용될 것입니다. 따라서, 먼저 이전의 루프 X_j 에서 계산된 보정값 C 를 빼서 수정된 피연산자 Y 를 유도합니다. 그 다음에, 이 피연산자를 전체 합 S 에 더하게 됩니다. Y 의 하위 비트(Y_l) 들은 합 계산시 사라지게 됩니다. 그 다음에, Y 의 상위 비트들을 $T - S$ 를 계산함으로써 구합니다. 만일 Y 가 여기서 빼진다면, Y 의 하위 비트들을 복구할 수 있겠지요. 이 값은 그림의 첫 번째 합에서 읽어버렸던 비트들이 됩니다. 이들은 다음 루프에서의 보정값이 됩니다.

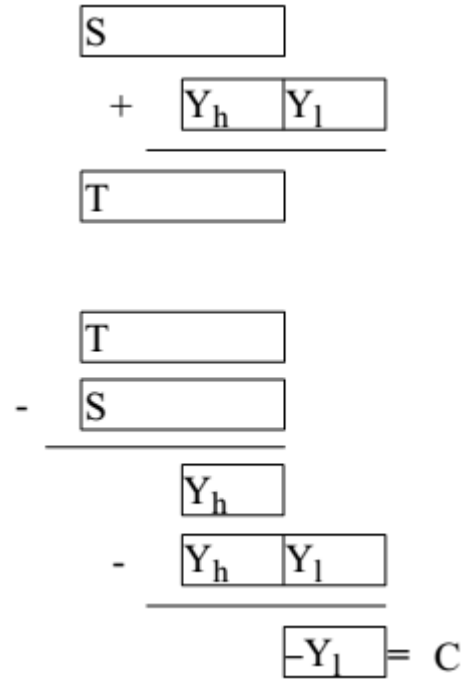

그림 4.1: Kahan 의 합 공식 진행 과정

제 5 장

끝마치며

컴퓨터 시스템 디자이너들이 부동 소수점의 시스템 관련한 부분들을 무시하는 것은 결코 흔하지 않은 일이 아닙니다. 이는 아마, 컴퓨터 과학 커리큘럼에서 부동 소수점에 대해 심층적으로 다루는 경우가 거의 없기 때문이죠.

이 문서에서는 부동 소수점이란 주제에 대해 매우 논리적으로 증명할 수 있다는 것을 보여주었습니다. 예를 들어서, 지워짐을 포함하고 있는 부동 소수점 알고리즘들은, 만일 하드웨어에서 보호 숫자를 도입한다면 매우 작은 상대 오차로 계산할 수 있다는 것이고, 이진–십진 변환 알고리즘의 경우 확장 정밀도가 제공되면 효율적으로 정확하게 작동한다는 사실 등입니다. 신뢰할 수 있는 부동 소수점 소프트웨어를 만드는 것은, 하드웨어 측면에서 여러가지 좋은 부동 소수점 기능들이 지원이 될 때 비로소 가능한 일입니다. 앞서 이야기 한 두 개의 예(보호 숫자와 확장 정밀도) 외에도, 시스템 적 측면 섹션에서 명령어 세트 디자인과 컴파일러 최적화에 까지 어떻게 하면 부동 소수점을 더 잘 지원할 수 있는지에 대해 설명하였습니다.

IEEE 부동 소수점 표준안이 점점 널리 채택되면서, 표준안을 지원하는 코드들은 이식성이 더욱 높아지고 있습니다. 섹션 IEEE 표준에서, 어떻게 하면 IEEE 표준의 기능들을 이용해서 실용적인 부동 소수점 코드를 작성할 수 있는지에 대해 이야기 하였습니다.

5.1 감사의 말

이 문서는 썬마이크로시스템 (Sun Microsystems) 사에서 1988년 5월 부터 7월 까지 이루어졌던 W.Kahan 의 강의에서 많은 영감을 받았습니다. 또한 이 강의를 열릴 수 있게Sun 사의 David Hough 씨가 많은 도움을 주셨습니다. 제 꿈은 일찍 일어나 아침 8 시에 진행되는 강의를 듣지 않고도, 사람들이 부동 소수점과 컴퓨터 시스템과의 상호 작용에 대해 공부할 수 있도록 하는 것입니다. Kahan 과 Xerox PARC 에 근무하는 저의 많은 동료들 (특히 John Gilbert) 에게 이 문서를 읽고 많은 유용한 코멘트를 달아준 것에 대해 감사를 표합니다. 또한 Paul Hilfinger 와 익명의 심사위원들에게 받았던 리뷰 역시 많은 도움이 되었습니다.