SAPIENZA
UNIVERSITÀ DI ROMA

Calcolo dei numeri reali esatti in Haskell

Facoltà di Ingegneria dell'informazione, informatica e statistica

Corso di Laurea in Informatica

Candidato

Andrea Princic
Matricola 1837592

Relatore

Prof. Ivano Salvo

Anno Accademico 2020/2021

Tesi non ancora discussa

Calcolo dei numeri reali esatti in Haskell
Tesi di Laurea. Sapienza – Università di Roma

Questa tesi è stata composta con LaTeX e la classe Sapthesis.

Versione: 31 ottobre 2021

Email dell'autore: princic.1837592@studenti.uniroma1.it

Sommario

Il tema della tesi è il calcolo dei numeri reali esatti, ovvero la possibilità di eseguire operazioni usando numeri decimali con un numero di cifre potenzialmente infinito e non periodico con precisione arbitraria. Questo è un problema importante in applicazioni pratiche che necessitano di fare calcoli molto precisi nei quali un piccolo errore di approssimazione potrebbe propagarsi e ingigantirsi nel susseguirsi dei calcoli, andando a impattare sulla precisione del risultato finale della computazione.

La computazione esatta fa uso di rappresentazioni e algoritmi tali che si possa raggiungere una precisione arbitraria prima, durante e dopo l'esecuzione di una serie di operazioni sui numeri reali. Queste rappresentazioni possono essere molto diverse da quelle usate comunemente nei calcolatori e fanno uso di strutture dati virtualmente infinite, chiamate stream. L'utilizzo di queste rappresentazioni e rispettivi algoritmi potrebbe comportare una perdita dal punto di vista dell'efficienza nell'esecuzione delle operazioni, ma offre il vantaggio di non avere perdite di precisione durante tutto il processo di calcolo

La tesi è suddivisa in 3 parti fondamentali:
introduzione al problema, con illustrazione di alcune possibili soluzioni;
illustrazione della rappresentazione e degli algoritmi scelti per questa tesi;
discussione dell'implementazione dei suddetti rappresentazione e algoritmi nel linguaggio di programmazione funzionale Haskell.

Indice

Capitolo 1

Computazione esatta sui numeri reali

1.1 Il problema

Il problema della rappresentazione dei numeri reali all'interno dei calcolatori elettronici è un problema che riguarda principalmente la precisione. Alcuni insiemi di numeri ($\mathbb{N}$, $\mathbb{Z}$) sono facilmente rappresentabili nel sistema binario come sequenze di bit. Questi, in base al numero di bit usati per rappresentarli, hanno un limite superiore di grandezza (e uno inferiore per i numeri con segno), ma rimangono comunque numeri precisi, senza approssimazioni. Non vale la stessa cosa per i numeri razionali ($\mathbb{Q}$) né tanto meno per i numeri reali ($\mathbb{R}$)

I numeri razionali vengono solitamente rappresentati con una rappresentazione chiamata mantissa-esponente, dal significato simile alla notazione scientifica, che rappresenta un numero come una coppia composta da:

- mantissa: una sequenza di bit, con segno positivo o negativo, la cui lunghezza determina la precisione del numero
- esponente: un numero intero che rappresenta l'effettiva grandezza del numero rappresentato

Usando questa rappresentazione abbiamo che una coppia mantissa-esponente $(m,\ e)$ nel sistema binario rappresenta il numero decimale

$$\frac{m}{2^{p-1}} \times 2^e$$

dove p è il numero di bit della mantissa. Questa rappresentazione può ovviamente essere usata con qualunque altra base diversa da 2.

I numeri razionali hanno la proprietà di poter essere rappresentati con una quantità finita di memoria. Più è grande il numero, più precisa dovrà essere la rappresentazione, e più memoria sarà necessaria per rappresentarlo ma, comunque, rimarrebbe sempre una quantità finita. Questo, però, non accade con i numeri reali non razionali. I numeri reali non sono rappresentabili con una quantità finita di memoria (o di cifre) in quanto la loro parte decimale è infinita (come anche quella

dei razionali) ma ha la particolarità di non essere periodica. Questo implica che per rappresentarli servirebbe una quantità infinita di memoria, una cosa impossibile sugli attuali calcolatori elettronici.

Dunque i numeri rappresentabili in un moderno calcolatore sono soltanto un sottoinsieme dei numeri razionali, la cui grandezza dipende dalla quantità di memoria destinata a rappresentarli. Qualunque numero che non appartenga a quel sottoinsieme rappresentabile viene necessariamente arrotondato al più vicino numero rappresentabile, causando un'inevitabile perdita di precisione sul risultato finale.

Un esempio di questa imprecisione è il risultato della somma tra i numeri razionali 0.1 e 0.2 fatta con il tipo float (32 bit):

```
0.1f + 0.2f = 0.30000001192092895507812 5
```

e con il tipo double (64 bit):

```
0.1d + 0.2d =
0.30000000000000004440892098500626161694526672363281 25
```

Oppure, provando ad usare il tipo float sul più grande valore rappresentabile da un intero a 32 bit: 2147483647 si ottiene invece il valore 2147483648 e facendovi alcune operazioni questi sono i risultati:

```
2147483647.0f = 2147483648.000000
2147483647.0f - 1 = 2147483648.000000
2147483647.0f - 64 = 2147483648.000000
2147483647.0f - 65 = 2147483520.000000
```

Questo accade perché a quella precisione il range che separa due float adiacenti è 128, quindi per ottenere un cambiamento bisogna sottrarre (o sommare) almeno la metà di questo range, passando così al float precedente (o successivo) [2].

Nei prossimi capitoli vengono esposte alcune rappresentazioni che permettono di risolvere questo problema.

1.2 Rappresentazioni per i reali

Come fare, dunque, per rappresentare un numero reale, o per lo meno un razionale a precisione arbitraria, in un calcolatore elettronico? Prima di tutto servirebbe una struttura dati virtualmente infinita, qualcosa che permetta di immagazzinare una quantità non finita di dati, ovviamente non tutti insieme (la memoria di un calcolatore è pure sempre finita). Una struttura che si presta a questo scopo è lo stream.

1.2.1 Gli stream

Gli stream non sono vere e proprie strutture dati. In generale, uno stream è una sequenza di elementi, i quali però non vengono elaborati tutti insieme ma soltanto

uno alla volta, quando è necessario. Gli stream quindi si prestano bene allo scopo di rappresentare sequenze potenzialmente infinite di cifre, dato che per calcolarne una parte non è necessario calcolarle per forza tutte.

Si potrebbe ad esempio ipotizzare che in uno stream infinito sia rappresentato un numero reale, con tutte le sue infinite cifre. Ora, anche se non se ne può ottenere il valore esatto, si può comunque ottenere una quantità arbitrariamente grande di cifre, tale da rendere il valore ottenuto preciso a tal punto da poterne trascurare l'inesattezza.

Gli algoritmi che utilizzano gli stream, non potendo utilizzare tutto il loro contenuto insieme, fanno tipicamente uso di alcune cifre di lookahead, ovvero utilizzano soltanto un numero finito di cifre dalla testa degli stream in input per generare una o più cifre dello stream in output, per poi continuare ricorsivamente a generare cifre utilizzando il resto dello stream. Questi algoritmi quindi funzionano in modo molto diverso dai classici algoritmi per le operazioni a cui siamo abituati, che iniziano facendo i conti dall'ultima cifra a destra e vanno verso sinistra. Gli algoritmi sugli stream non possono accedere all'ultima cifra (in quanto non esiste un'ultima cifra) quindi eseguono i calcoli a partire da sinistra e procedono verso destra.

Nei prossimi capitoli si useranno le seguenti sintassi per riferirsi ad uno stream:

$$[x_1, x_2, x_3, \ldots] \equiv x_1 : x_2 : x_3 : \ldots$$

La seguente sintassi per riferirsi alla concatenazione di cifre in testa ad uno stream, dove x è uno stream e x_i è una cifra:

$$x_1 : x_2 : x_3 : x$$

La seguente sintassi per riferirsi ad uno stream infinito della cifra x

$$\overrightarrow{x}$$

Inoltre va fatta una distinzione tra il concetto di numero e di numerale: un numero è il valore matematico derivato dall'interpretazione di un numerale, il quale nel nostro caso sarà uno stream o più avanti una coppia simile a quella usata nella rappresentazione mantissa-esponente.
Per differenziare questi due concetti si usano le doppie parentesi quadre $[\![x]\!]$ per indicare il numero rappresentato dal numerale x.

Un'ultima cosa da notare è che, nella maggior parte dei casi, nei prossimi capitoli si useranno gli stream per rappresentare soltanto un intervallo chiuso di numeri reali e non l'intera linea, visto che negli stream manca l'informazione che servirebbe a distinguere la parte intera dalla parte decimale di un numero. In generale, uno stream

$$x = d_1 : d_2 : d_3 : \ldots$$

in base b verrà interpretato come

$$[\![x]\!] = \sum_{i=1}^{\infty} d_i \times b^{-i} = \frac{d_1}{b} + \frac{d_2}{b^2} + \frac{d_3}{b^3} + \ldots$$

1.2.2 Stream di cifre

Una prima e semplice rappresentazione dei numeri reali tramite stream potrebbe essere quella di rappresentarli come semplici sequenze di cifre, decimali o binarie che sia, da sinistra a destra. Questa rappresentazione, purtroppo, non può funzionare. Si prenda ad esempio il caso dei numeri $0.333\ldots$ e $0.666\ldots$ in base 10, rappresentati dai seguenti stream:

$$3 : 3 : 3 : \ldots \qquad 6 : 6 : 6 : \ldots$$

e si ipotizzi di volerne fare la somma.
Tale operazione non sarebbe computabile su questi due numeri in questa rappresentazione perché per determinare anche soltanto la prima cifra del risultato servirebbe una quantità infinita di input letto dai due addendi.
Questo accade per via del fatto che uno stream che inizia con una lunga sequenza di 3 non per forza rappresenta il numero $\frac{1}{3}$, e allo stesso modo uno stream che inizia con una lunga sequenza di 6 non per forza rappresenta il numero $\frac{2}{3}$. Potrebbe essere che ad un certo punto di quegli stream, magari dopo migliaia di cifre, si trovi una cifra diversa dal 3 o dal 6, ad esempio la cifra 0. In tal caso il risultato della somma non sarebbe 1 ma un numero che si trova nell'intervallo

$$[0.9, 1)$$

ma per stabilirlo servirebbe analizzare una quantità potenzialmente infinita di cifre da entrambi gli stream.

Lo stesso problema sorge anche nel caso di basi diverse da 10:
ad esempio in base 2 si prendano in esame gli stream

$$1 : 1 : 1 : \ldots \qquad \text{e} \qquad 0 : 0 : 0 : \ldots$$

che potrebbero rappresentare i numeri 1 e 0, ma non è possibile saperlo senza esaminarne potenzialmente tutte le cifre.

Questa rappresentazione è quindi da escludere dalle possibili soluzioni.

1.2.3 Cifre binarie con segno

La seconda rappresentazione [3] è simile alla prima (ora si parla di base 2) ma aggiunge una nuova cifra alle due usate normalmente: la cifra -1 alla quale successivamente, nella sintassi degli stream, mi riferirò usando il simbolo $\overline{1}$.

La cifra $\overline{1}$ funziona esattamente come le altre cifre, soltanto che il suo contributo al valore del numerale è negativo. Questo quindi amplia l'insieme dei numeri rappresentabili tramite stream non più all'intervallo $[0, 1]$ ma all'intervallo $[-1, 1]$.

L'uso di questa cifra comporta anche l'introduzione di una ridondanza in questa notazione. Ridondanza significa che alcuni numeri possono essere scritti non più in un unico modo, ma in molti modi diversi.
In questa notazione, infatti, gli unici due numeri che mantengono una rappresentazione unica sono

$$1 = \left[\!\left[\overrightarrow{1} \right]\!\right] \qquad \text{e} \qquad \overline{1} = \left[\!\left[\overrightarrow{\overline{1}} \right]\!\right]$$

Ad esempio il numero 0 può essere espresso come

$$0 = \left[\!\left[\overrightarrow{0}\right]\!\right] \qquad \text{oppure} \qquad 0 = \left[\!\left[1 : \overrightarrow{\overline{1}}\right]\!\right] \qquad \text{oppure} \qquad 0 = \left[\!\left[\overline{1} : \overrightarrow{1}\right]\!\right]$$

Infatti:

$$\left[\!\left[1 : \overrightarrow{\overline{1}}\right]\!\right] = \frac{1}{2} + \sum_{i=2}^{\infty} \overline{1} \times 2^{-i} = \frac{1}{2} - \frac{1}{4} - \frac{1}{8} - \frac{1}{16} - \ldots = 0$$

$$\left[\!\left[\overline{1} : \overrightarrow{1}\right]\!\right] = -\frac{1}{2} + \sum_{i=2}^{\infty} 1 \times 2^{-i} = -\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \ldots = 0$$

Va inoltre fatto notare che in questa rappresentazione esistono anche le seguenti identità:

$$1 : \overline{1} : x = 0 : 1 : x$$
$$\overline{1} : 1 : x = 0 : \overline{1} : x$$

Uno dei vantaggi di questa rappresentazione è che permette di eseguire anche quei calcoli che nella rappresentazione classica necessiterebbero di una quantità infinita di input per produrre anche solo una cifra di output. Questo succede perché grazie alla cifra $\overline{1}$ si possono aggiustare eventuali errori di eccesso avvenuti durante i precedenti passi del calcolo. Questi errori in questa rappresentazione sono comuni in quanto, come detto in precedenza, gli algoritmi sugli stream devono produrre cifre a partire da sinistra e non da destra. Questo porta spesso alla produzione di una cifra, ad esempio 1, che però potrebbe risultare più avanti una cifra troppo grande in quella posizione. Questo errore, che non si potrebbe correggere nella rappresentazione classica, si può invece aggiustare in questa rappresentazione grazie alla cifra $\overline{1}$, che permette di diminuire il valore del numero in output.

Tornando al caso della somma tra gli stream

$$1 : 1 : 1 : \ldots \qquad \text{e} \qquad 0 : 0 : 0 : \ldots$$

che non poteva essere computata nella rappresentazione classica: il problema stava nel fatto che la prima cifra di questa somma può essere:

- 1 nel caso in cui il valore della somma dovesse risultare ≥ 1
- 0 altrimenti

soltanto che per saperlo si sarebbe dovuta analizzare una parte potenzialmente infinita di input. Questo problema non c'è più a seguito dell'introduzione della cifra $\overline{1}$ in quanto, anche dando in output 1 come prima cifra, nel caso in cui il risultato reale si rivelasse più piccolo di quello atteso, si possono dare in output una o più cifre $\overline{1}$ per ridurre il valore del risultato calcolato e farlo arrivare al giusto valore della somma.

Alcune proprietà dei numeri binari vengono mantenute in questa rappresentazione mentre altre vengono migliorate. Ad esempio in questa rappresentazione si possono facilmente rappresentare numeri negativi e si può passare da un numero al suo opposto semplicemente invertendone le cifre:

$$\left[\!\left[1 : \overline{1} : 0 : 1 : 0 : \overrightarrow{0}\right]\!\right] = \quad 0.3125$$
$$\left[\!\left[\overline{1} : 1 : 0 : \overline{1} : 0 : \overrightarrow{0}\right]\!\right] = -0.3125$$

Anche le proprietà della divisione e moltiplicazione per 2 tramite shift valgono, con l'accortezza che i numeri in questa rappresentazione possono trovarsi soltanto nell'intervallo $[-1, 1]$, e quindi non è sempre detto che si possa moltiplicare per 2 un numero senza causare overflow:

$$\begin{aligned}
\left[\!\left[1 : \overline{1} : 0 : 1 : 0 : \overrightarrow{0}\right]\!\right] &= \quad 0.3125 \\
\left[\!\left[0 : 1 : \overline{1} : 0 : 1 : \overrightarrow{0}\right]\!\right] &= \quad 0.15625 = \frac{0.3125}{2} \\
\left[\!\left[\overline{1} : 0 : 1 : 0 : 0 : \overrightarrow{0}\right]\!\right] &= -0.375 \quad \neq 0.3125 \cdot 2 = 0.625
\end{aligned}$$

È possibile però notare che, anche se il numero 0.3125 è minore di 0.5, la sua moltiplicazione causa comunque un overflow, nonostante il numero

$$0.3125 \cdot 2 \;=\; 0.625$$

sia rappresentabile nel range $[-1, 1]$. Questo succede a causa della ridondanza introdotta dalla cifra $\overline{1}$. Possiamo notare infatti che le prime due cifre dello stream

$$1 : \overline{1} : 0 : 1 : 0 : \overrightarrow{0}$$

possono essere cambiate in

$$0 : 1 : 0 : 1 : 0 : \overrightarrow{0}$$

grazie alle due uguaglianze indicate in precedenza. Ora, con questa nuova forma dello stesso numero, ci è possibile eseguire la moltiplicazione per 2 tramite shift ottenendo il risultato corretto:

$$\left[\!\left[1 : 0 : 1 : 0 : \overrightarrow{0}\right]\!\right] = 0.625$$

Gli algoritmi relativi a questa rappresentazione utilizzano queste uguaglianze per manipolare gli stream in input e risolvere questi errori.

Per quanto riguarda il calcolo di numeri reali sull'intera linea e non soltanto nell'intervallo $[-1, 1]$, si usa una rappresentazione simile a mantissa-esponente:

$$[\![x]\!] = [\![(m, e)]\!] = [\![m]\!] \times 2^e$$

dove x è un numerale, m è uno stream di cifre binarie con segno ed e è un numero intero di grandezza arbitraria

1.2.4 Rappresentazione in base ϕ

Un'alternativa all'introduzione di cifre con peso negativo è quella di cambiare completamente base, usando un numero irrazionale invece di un numero naturale: il numero ϕ [1].

$$\phi = \frac{\sqrt{5} + 1}{2}$$

L'utilizzo del numero ϕ come base rende possibile la rappresentazione e il calcolo dei numeri reali usando solamente due cifre: 0 e 1. I numeri in base ϕ, dunque,

appaiono come dei semplici numeri binari, soltanto che il peso di ogni cifra non è una potenza di 2 ma una potenza di ϕ.

Questa base, come la rappresentazione con cifre negative, presenta una grande ridondanza. Esiste infatti la seguente identità:

$$1.00 = 0.11$$

E questo vale in qualunque posizione si trovino le cifre $1, 0, 0$. Ad esempio:

$$10.0 = 1.1 \qquad (\phi)$$
$$100 = 11 \qquad (\phi^2)$$
$$0.1100 = 0.1011 \qquad (1)$$

Questo è vero per via del fatto che:

$$\phi^n + \phi^{n+1} = \phi^{n+2}$$

L'utilizzo di una base irrazionale comporta anche un importante cambiamento nella forma dei numeri rappresentati. Basta pensare, ad esempio, che lo stesso numero ϕ, che in qualunque base naturale è un numero irrazionale (ovvero non si può rappresentare con un numero finito di cifre), in questa base diventa un semplice numero naturale: 10. Infatti:

$$10_\phi = 1 \cdot \phi^1 \ + \ 0 \cdot \phi^0 = \phi$$

Non è vero, invece, il contrario: ogni numero naturale in base ϕ può infatti essere rappresentato con un numero finito di cifre. Ad esempio:

$$1_{10} = 1 = 0.11$$

$$2_{10} = 1 + 0.11 = 1.11 = 10.01$$

$$3_{10} = 10.01 + 1 = 11.01 = 100.01$$

$$4_{10} = 100.01 + 1 = 101.01 = 101.0011$$

$$5_{10} = 101.0011 + 0.11 = 101.1111 = 110.0111$$

Per quanto riguarda la rappresentazione dei numeri reali in questa base, anche in questo caso si usano stream infiniti di cifre. Questa volta, però, i numeri rappresentabili tramite questi stream si trovano nell'intervallo $[0, \phi]$. In questa rappresentazione gli unici numeri che si possono scrivere in un unico modo sono

$$\phi = \overrightarrow{1} \qquad \text{e} \qquad 0 = \overrightarrow{0}$$

mentre tutti gli altri sono rappresentabili in infiniti modi grazie all'identità tipica di questa base.

Numeri di Fibonacci

Una caratteristica di questa base è la facilità con cui si possono rappresentare e calcolare i numeri di Fibonacci. In generale, i numeri di Fibonacci sono dati dalla seguente funzione ricorsiva:

$$\begin{cases} Fib(1) = 1 \\ Fib(2) = 1 \\ Fib(n) = Fib(n-1) + Fib(n-2) \end{cases}$$

oppure dalla formula di Binet:

$$Fib_n = \frac{\phi^n}{\sqrt{5}} - \frac{(1-\phi)^n}{\sqrt{5}} = \frac{\phi^n - (-\phi)^{-n}}{\sqrt{5}}$$

Queste due formule funzionano indipendentemente dalla base che si usa, ma la rappresentazione in base ϕ si caratterizza perché in questa base i numeri di Fibonacci seguono un pattern fatto da una cifra 1 seguita da tre cifre 0, e inoltre i numeri che occupano una posizione pari nella sequenza di Fibonacci si distinguono da quelli che occupano una posizione dispari in base al modo in cui finiscono. Il pattern e la differenza tra posizioni pari e dispari sono facilmente individuabili in questa porzione di sequenza di Fibonacci:

1	=	0.11	= 1	(1)
1	=	1	= 1	(2)
10.01	=	10.0011	= 2	(3)
100.01	=	100.01	= 3	(4)
1000.1001	=	1000.100011	= 5	(5)
10001.0001	=	10001.0001	= 8	(6)
100010.001001	=	100010.00100011	= 13	(7)
1000100.010001	=	1000100.010001	= 21	(8)
10001000.10001001	=	10001000.1000100011	= 34	(9)
100010001.00010001	=	100010001.00010001	= 55	(10)
1000100010.0010001001	=	1000100010.001000100011	= 89	(11)
10001000100.0100010001	=	10001000100.0100010001	= 144	(12)

Da questi esempi si può facilmente notare che il passaggio da un numero al successivo viene fatto principalmente con uno shift a sinistra e, in base alla posizione nella sequenza (pari o dispari), con l'aggiunta o la rimozione di alcune cifre a destra del numero.

Di seguito la dimostrazione che il pattern vale per ogni numero di Fibonacci in base ϕ, fatta per induzione e basandosi sul fatto che il pattern è diverso per numeri in posizione pari e dispari.

Teorema 1. *Un qualunque numero di Fibonacci $Fib(n)$ con $n > 2$ può essere ottenuto dal precedente nel seguente modo:*

Se n è dispari allora $Fib(n)$ si ottiene facendo uno shift a sinistra delle cifre di $Fib(n-1)$ e aggiungendo le cifre 00011 *a destra.*
Se n è pari allora $Fib(n)$ si ottiene facendo uno shift a sinistra delle cifre di $Fib(n-1)$ e rimuovendo l'ultima cifra 1 *a destra.*

Dimostrazione. I numeri di Fibonacci $Fib(n)$ con $n \leq 12$ soddisfano la proprietà, come mostrato nella tabella in alto. Per i numeri con $n > 12$ si procede come segue:

Se n è dispari il numero si deve ottenere tramite shift a sinistra e aggiunta di cifre 00011 a destra.
Supponiamo che i numeri fino a $Fib(n-1)$ soddisfino il pattern. Allora i numeri $Fib(n-1)$ e $Fib(n-2)$ hanno la seguente forma:

$$\begin{aligned} 100010\ldots00100011 \quad & (n-2) \\ 1000100\ldots010001 \quad & (n-1) \end{aligned}$$

Dalla cui somma si ottiene il numero:

$$1100110\ldots01100111 = 10001000\ldots1000100011$$

Che messo in colonna con i due precedenti mostra come ci sia stato uno shift a sinistra e aggiunta di cifre 00011 a destra rispetto al precedente:

$$\begin{aligned} 100010\ldots00100011 \quad & (n-2) \\ 1000100\ldots010001 \quad & (n-1) \\ 10001000\ldots1000100011 \quad & (n) \end{aligned}$$

Se n è pari il numero si deve ottenere tramite shift a sinistra e rimozione dell'ultima cifra 1 a destra.
Supponiamo che i numeri fino a $Fib(n-1)$ soddisfino il pattern. Allora i numeri $Fib(n-1)$ e $Fib(n-2)$ hanno la seguente forma:

$$\begin{aligned} 100010\ldots0010001 \quad & (n-2) \\ 1000100\ldots01000100011 \quad & (n-1) \end{aligned}$$

Dalla cui somma otteniamo il numero:

$$1100110\ldots01100110011 = 10001000\ldots100010001$$

Che messo in colonna con i due precedenti mostra come ci sia stato uno shift a sinistra e rimozione dell'ultima cifra 1 rispetto al precedente:

$$\begin{aligned} 100010\ldots0010001 \quad & (n-2) \\ 1000100\ldots01000100011 \quad & (n-1) \\ 10001000\ldots100010001 \quad & (n) \end{aligned}$$

□

Questa proprietà è causata dall'identità dei numeri in base ϕ:

$$1.00 = 0.11$$

che provoca uno slittamento a sinistra di due cifre 1 consecutive. Nel caso dei numeri di Fibonacci, visto che ogni numero è ottenuto dal precedente facendo uno shift a sinistra, ogni somma di due numeri di Fibonacci consecutivi si trova ad avere una coppia di 1 dove nei precedenti c'erano le singole cifre 1, e questo provoca lo shift a sinistra.

Usando questa proprietà si può ottenere facilmente un numero di Fibonacci arbitrario senza eseguire nessuna operazione ricorsiva ma sfruttando solamente la posizione nella sequenza che si vuole calcolare.
In alternativa, volendo sfruttare la ricorsione, si può calcolare un numero di Fibonacci arbitrario senza eseguire nessuna operazione matematica sul numero stesso, ma usando soltanto la concatenazione di cifre ai risultati ottenuti ricorsivamente.
Entrambe queste funzioni sono state implementate in Haskell.

Capitolo 2

La Golden Notation

La rappresentazione in base ϕ, dunque, permette di rappresentare numeri nell'intervallo $[0, \phi]$. Questa notazione verrà d'ora in avanti chiamata notazione semplificata, e la sua interpretazione da numerale a numero verrà fatta così [1]:

$$[\![\alpha]\!]_s = \sum_{i=1}^{\infty} \alpha_i \times \phi^{-i}$$

Volendo espandere la rappresentazione in base ϕ a tutta la linea dei reali, si può usare anche in questo caso una rappresentazione simile a mantissa-esponente, ma c'è un problema: l'utilizzo di stream infiniti in base ϕ non permette rappresentare i numeri negativi. Per rimediare a questa mancanza si potrebbe pensare di ampliare la rappresentazione mantissa-esponente a una rappresentazione del tipo segno-mantissa-esponente. Questo purtroppo non è possibile in quanto, analogamente al problema della somma

$$3 : 3 : 3 : \ldots \; + \; 6 : 6 : 6 : \ldots$$

per decidere il segno di un numero servirebbe sapere se è minore o maggiore di zero, ma questo non è possibile quando si lavora con stream infiniti.
La soluzione è quindi di modificare il modo in cui una coppia mantissa-esponente viene interpretata, aggiungendo un termine che permette di rappresentare numeri negativi. Questa notazione verrà d'ora in avanti chiamata notazione completa, e la sua interpretazione da numerale a numero verrà fatta così:

$$[\![z : \alpha]\!]_f = (-1 + [\![\alpha]\!]_s) \times \phi^{2z} = \left(-1 + \sum_{i=1}^{\infty} \alpha_i \times \phi^{-i}\right) \times \phi^{2z}$$

dove α è uno stream nella notazione semplificata e z è un intero di grandezza arbitraria. Nella notazione completa un numerale è uno stream il cui primo elemento è l'esponente, ma questa notazione ha un significato del tutto equivalente alla coppia $(z,\ \alpha)$.
Il termine -1 viene usato per separare i valori di uno stream nella notazione semplificata tra positivi e negativi. In questo modo tutti i valori nell'intervallo $[0, 1)$ diventano negativi mentre i valori nell'intervallo $(1, \phi]$ rimangono positivi. Il valore 1 nella notazione semplificata diventa 0 nella notazione completa, indipendentemente dall'esponente.

Gli algoritmi sono divisi tra notazione semplificata e notazione completa. Quelli per la notazione semplificata, non essendo chiusi sull'intervallo $[0, \phi]$, restituiscono un valore diviso per un coefficiente di ϕ oppure ϕ^2 in base a quanti bit di overflow possono essere generati dall'algoritmo. Quelli per la notazione completa fanno uso di quelli per la notazione semplificata.

Come detto in precedenza, gli algoritmi sugli stream lavorano da sinistra a destra utilizzando un certo numero di cifre di lookahead dalla testa dello stream e fanno uso delle identità tipiche della rappresentazione per modificare il contenuto di uno stream senza cambiarne il valore.

2.1 Addizione semplificata

L'algoritmo per la notazione semplificata fa uso di un massimo di due cifre di lookahead e utilizza due cifre di riporto con peso relativamente $\frac{d_1}{\phi}$ e $\frac{d_2}{\phi^2}$. Il suo risultato è diviso per ϕ^2 per far rientrare due eventuali cifre di overflow. Per l'addizione nella notazione semplificata vale dunque la seguente uguaglianza:

$$[\![A(\alpha,\ \beta,\ a,\ b)]\!]_s = \frac{[\![\alpha]\!]_s + [\![\beta]\!]_s + \frac{a}{\phi} + \frac{b}{\phi^2}}{\phi^2}$$

Le regole basate sulle cifre di lookahead sono:

$$\begin{aligned}
A(0:\alpha,\ 0:\beta,\ 0,\ b) &= 0 : A(\alpha,\ \beta,\ b,\ 0)\\
A(0:0:\alpha,\ 0:\beta,\ 1,\ b) &= 0 : A(b:\alpha,\ \beta,\ 1,\ 1)\\
A(0:1:\alpha,\ 0:1:\beta,\ 1,\ 1) &= 1:0 : A(\alpha,\ \beta,\ 0,\ 1)\\
A(0:0:\alpha,\ 1:0:\beta,\ 1,\ 0) &= 0:1 : A(\alpha,\ \beta,\ 1,\ 0)\\
A(0:\alpha,\ 1:\beta,\ 1,\ 1) &= 1 : A(\alpha,\ \beta,\ 0,\ 0)\\
A(1:\alpha,\ 1:\beta,\ 1,\ b) &= 1 : A(\alpha,\ \beta,\ b,\ 1)
\end{aligned}$$

$$\begin{aligned}
A(1:\alpha,\ 0:\beta,\ a,\ b) &= A(0:\alpha,\ 1:\beta,\ a,\ b)\\
A(\alpha,\ 1:\beta,\ 0,\ b) &= A(\alpha,\ 0:\beta,\ 1,\ b)\\
A(a_1:1:\alpha,\ b_1:0:\beta,\ a,\ b) &= A(a_1:0:\alpha,\ b_1:1:\beta,\ a,\ b)\\
A(\alpha,\ b_1:1:\beta,\ a,\ 0) &= A(\alpha,\ b_1:0:\beta,\ a,\ 1)
\end{aligned}$$

2.2 Addizione completa

L'algoritmo per l'addizione completa si basa su quello per l'addizione semplificata e su operazioni sull'esponente, facendo distinzione in base alla relazione che c'è tra gli esponenti degli addendi:

$$A'(z:\alpha,\ t:\beta) = \begin{cases} (z+1) : A(\alpha,\ \beta,\ 1,\ 0) & \text{se } z = t\\ A'((z+1):1:0:\alpha,\ t:\beta) & \text{se } z < t\\ A'(z:\alpha,\ (t+1):1:0:\beta) & \text{se } t < z \end{cases}$$

L'algoritmo esegue la vera operazione solo nel caso in cui i due esponenti siano uguali. Negli altri casi non fa altro che aumentare il minore dei due esponenti fino a portarli ad una situazione di parità, sfruttando la seguente uguaglianza:

$$\begin{aligned}
[\![(z+1):1:0:\alpha]\!]_f &= (-1 + [\![1:0:\alpha]\!]_s) \times \phi^{2z+2} \\
&= \left(-1 + \frac{1}{\phi} + \frac{[\![\alpha]\!]_s}{\phi^2}\right) \times \phi^{2z+2} \\
&= \left(-\phi^2 + \phi + [\![\alpha]\!]_s\right) \times \phi^{2z} \\
&= (-1 + [\![\alpha]\!]_s) \times \phi^{2z} \\
&= [\![z:\alpha]\!]_f
\end{aligned}$$

2.3 Complemento semplificato

Per quanto riguarda la sottrazione, non si implementa un vero algoritmo per sottrarre due numeri in notazione semplificata in quanto l'intervallo $[0, \phi]$ non include i numeri negativi. Si utilizza invece un algoritmo che nega tutte le cifre di uno stream. Il risultato della negazione di uno stream in notazione semplificata è:

$$[\![C(\alpha)]\!]_s = \phi - [\![\alpha]\!]_s$$

Questo deriva dal fatto che:

$$[\![\alpha]\!]_s + [\![C(\alpha)]\!]_s = \left[\!\left[\overrightarrow{1}\right]\!\right]_s = \phi$$

Dove:

$$\begin{aligned}
C(1:\alpha) &= 0 : C(\alpha) \\
C(0:\alpha) &= 1 : C(\alpha)
\end{aligned}$$

2.4 Complemento e Sottrazione completi

In questa sezione si illustrano gli algoritmi per complemento e sottrazione nella notazione completa. È importante notare che l'algoritmo per la sottrazione non fa uso dell'algoritmo per il complemento, il quale però verrà usato in seguito per moltiplicazione e divisione.

La funzione per effettuare il complemento di un numero nella notazione completa fa uso di una funzione ausiliaria che si basa sulla funzione C della notazione semplificata:

$$\begin{aligned}
C'(z:\alpha) &= (z+1) : C_1(\alpha) \\
C_1(0:\alpha) &= 1:1:0:C(\alpha) \\
C_1(1:0:\alpha) &= 1:0:C_1(\alpha) \\
C_1(1:1:\alpha) &= 1:0:0:1:C(\alpha)
\end{aligned}$$

L'algoritmo per la sottrazione completa diretta funziona in modo simile a quello dell'addizione completa e si basa sugli algoritmi per addizione e complemento semplificati:

$$S'(z:\alpha,\ t:\beta) = \begin{cases} (z+1) : A(\alpha, C(\beta), 1, 1) & \text{se } z = t \\ S'((z+1) : 1 : 0 : \alpha,\ t : \beta) & \text{se } z < t \\ S'(z : \alpha,\ (t+1) : 1 : 0 : \beta) & \text{se } t < z \end{cases}$$

Anche questo algoritmo effettua la vera operazione solo nel caso in cui i due esponenti siano uguali mentre negli altri casi non fa altro che incrementare il minore fino a pareggiarli, sfruttando la stessa uguaglianza usata nell'addizione completa.

2.5 Prodotto semplificato

L'algoritmo del prodotto nella notazione semplificata ha un funzionamento simile a quello dell'addizione: sfrutta un massimo di due cifre di lookahead e restituisce il risultato considerando due eventuali bit di overflow. I casi con le due cifre di lookahead sono quattro: uno per ogni combinazione di due bit, considerando che il primo bit è sempre 1. Al suo interno utilizza l'algoritmo dell'addizione per spezzare il prodotto in una serie di somme:

$$\begin{aligned}
P(0:\alpha,\ \beta) &= 0 : P(\alpha,\ \beta) \\
P(\alpha,\ 0:\beta) &= 0 : P(\alpha,\ \beta) \\
P(1:0:\alpha,\ 1:0:\beta) &= 0 : A(A(\alpha,\ \beta,\ 0,\ 0),\ 0 : P(\alpha,\ \beta),\ 1,\ 0) \\
P(1:1:\alpha,\ 1:0:\beta) &= A(A(0:\alpha,\ \beta,\ 0,\ 0),\ 0:0:P(\alpha,\ \beta),\ 1,\ 0) \\
P(1:0:\alpha,\ 1:1:\beta) &= A(A(\alpha,\ 0:\beta,\ 0,\ 0),\ 0:0:P(\alpha,\ \beta),\ 1,\ 0) \\
P(1:1:\alpha,\ 1:1:\beta) &= A(A(\alpha,\ \beta,\ 0,\ 0),\ 0:0:P(\alpha,\ \beta),\ 1,\ 1)
\end{aligned}$$

Il risultato del prodotto semplificato è quindi:

$$[\![P(\alpha,\ \beta)]\!]_s = \frac{[\![\alpha]\!]_s \times [\![\beta]\!]_s}{\phi^2}$$

2.6 Prodotto completo

L'algoritmo per il prodotto completo fa uso di quello per l'addizione, il prodotto e il complemento semplificati e fa la somma degli esponenti:

$$P'(z:\alpha,\ t:\beta) = (z+t+2) : A(P(\alpha,\ \beta),\ C(A(\alpha,\ \beta,\ 0,\ 0)),\ 1,\ 0)$$

La correttezza della formula:

$$[\![P'(z:\alpha,\ t:\beta)]\!]_f = [\![z:\alpha]\!]_f \times [\![t:\beta]\!]_f$$

deriva dalle seguenti uguaglianze:

$$
\begin{aligned}
&[\![z:\alpha]\!]_f \times [\![t:\beta]\!]_f && (2.1)\\
=&(-1+[\![\alpha]\!]_s)\times(-1+[\![\beta]\!]_s)\times\phi^{2z+2t} && (2.2)\\
=&(-1+[\![\alpha]\!]_s\times[\![\beta]\!]_s-[\![\alpha]\!]_s-[\![\beta]\!]_s)\times\phi^{2z+2t} && (2.3)\\
=&((-\phi^3-\phi-1)+\phi^3+\phi+[\![\alpha]\!]_s\times[\![\beta]\!]_s-[\![\alpha]\!]_s-[\![\beta]\!]_s)\times\phi^{2z+2t} && (2.4)\\
=&(-\phi^4+\phi^3+\phi+[\![\alpha]\!]_s\times[\![\beta]\!]_s-[\![\alpha]\!]_s-[\![\beta]\!]_s)\times\phi^{2z+2t} && (2.5)\\
=&(-\phi^4+[\![\alpha]\!]_s\times[\![\beta]\!]_s+\phi^3-[\![\alpha]\!]_s-[\![\beta]\!]_s+\phi)\times\phi^{2z+2t} && (2.6)\\
=&\left(-1+\frac{\frac{[\![\alpha]\!]_s\times[\![\beta]\!]_s}{\phi^2}+\phi-\frac{[\![\alpha]\!]_s+[\![\beta]\!]_s}{\phi^2}+\frac{1}{\phi}}{\phi^2}\right)\times\phi^{2z+2t+4} && (2.7)\\
=&\left(-1+\frac{[\![P(\alpha,\ \beta)]\!]_s+[\![C(A(\alpha,\ \beta,\ 0,\ 0))]\!]_s+\frac{1}{\phi}}{\phi^2}\right)\times\phi^{2z+2t+4} && (2.8)\\
=&(-1+[\![A(P(\alpha,\ \beta),\ C(A(\alpha,\ \beta,\ 0,\ 0)),\ 1,\ 0)]\!]_s)\times\phi^{2z+2t+4} && (2.9)\\
=&[\![(z+t+2):A(P(\alpha,\ \beta),\ C(A(\alpha,\ \beta,\ 0,\ 0)),\ 1,\ 0)]\!]_f && (2.10)\\
=&\left[\!\left[P'(z:\alpha,\ t:\beta)\right]\!\right]_f && (2.11)
\end{aligned}
$$

Un'importante precisazione va fatta su questo algoritmo: la dimostrazione di correttezza è sbagliata, dunque l'algoritmo stesso è sbagliato. L'errore è dovuto al passaggio da (2.2) a (2.3), in cui alla moltiplicazione -1×-1 è stato sostituito come risultato -1 invece di $+1$. Questo passaggio rende inesatti tutti i passaggi successivi e provoca un enorme errore nel risultato dell'algoritmo. All'atto pratico, infatti, l'utilizzo dell'algoritmo produce risultati che differiscono di migliaia di volte dal risultato corretto e tal volta di segno opposto.

2.6.1 Prodotto completo corretto

Fortunatamente la soluzione finale non è troppo lontana dalla soluzione corretta e la logica utilizzata nei passaggi intermedi è giusta, eccetto che per il segno dei termini. Di seguito è illustrato il procedimento corretto che conduce ad un algoritmo

funzionante:

$$
\begin{aligned}
&\llbracket z:\alpha \rrbracket_f \times \llbracket t:\beta \rrbracket_f \\
=&(-1+\llbracket \alpha \rrbracket_s) \times (-1+\llbracket \beta \rrbracket_s) \times \phi^{2z+2t} \\
=&(1+\llbracket \alpha \rrbracket_s \times \llbracket \beta \rrbracket_s - \llbracket \alpha \rrbracket_s - \llbracket \beta \rrbracket_s) \times \phi^{2z+2t} \\
=&((\phi^3+\phi+1) - \phi^3 - \phi + \llbracket \alpha \rrbracket_s \times \llbracket \beta \rrbracket_s - \llbracket \alpha \rrbracket_s - \llbracket \beta \rrbracket_s) \times \phi^{2z+2t} \\
=&-(-\phi^4+\phi^3+\phi - \llbracket \alpha \rrbracket_s \times \llbracket \beta \rrbracket_s + \llbracket \alpha \rrbracket_s + \llbracket \beta \rrbracket_s) \times \phi^{2z+2t} \\
=&-(-\phi^4+\phi^3 - \llbracket \alpha \rrbracket_s \times \llbracket \beta \rrbracket_s + \llbracket \alpha \rrbracket_s + \llbracket \beta \rrbracket_s + \phi) \times \phi^{2z+2t} \\
\\
=&-\left(-1+\frac{\phi - \frac{\llbracket \alpha \rrbracket_s \times \llbracket \beta \rrbracket_s}{\phi^2} + \frac{\llbracket \alpha \rrbracket_s + \llbracket \beta \rrbracket_s}{\phi^2} + \frac{1}{\phi}}{\phi^2}\right) \times \phi^{2z+2t+4} \\
\\
=&-\left(-1+\frac{\llbracket C(P(\alpha,\ \beta)) \rrbracket_s + \llbracket A(\alpha,\ \beta,\ 0,\ 0) \rrbracket_s + \frac{1}{\phi}}{\phi^2}\right) \times \phi^{2z+2t+4} \\
\\
=&-(-1+\llbracket A(C(P(\alpha,\ \beta)),\ A(\alpha,\ \beta,\ 0,\ 0),\ 1,\ 0) \rrbracket_s) \times \phi^{2z+2t+4} \\
=&-\llbracket (z+t+2) : A(C(P(\alpha,\ \beta)),\ A(\alpha,\ \beta,\ 0,\ 0),\ 1,\ 0) \rrbracket_f \\
=&\left\llbracket C'((z+t+2) : A(C(P(\alpha,\ \beta)),\ A(\alpha,\ \beta,\ 0,\ 0),\ 1,\ 0)) \right\rrbracket_f \\
=&\left\llbracket P'(z:\alpha,\ t:\beta) \right\rrbracket_f
\end{aligned}
$$

Da cui si ottiene l'algoritmo corretto per il prodotto completo:

$$P'(z:\alpha,\ t:\beta) = C'((z+t+2) : A(C(P(\alpha,\ \beta)),\ A(\alpha,\ \beta,\ 0,\ 0),\ 1,\ 0))$$

2.7 Divisione semplificata

L'algoritmo per la divisione semplificata è il più complicato: utilizza fino a tre cifre di lookahead, due funzioni ausiliarie mutuamente ricorsive ed ha una restrizione sul divisore, il quale deve per forza iniziare con un 1 per assicurarsi che il risultato sia abbastanza piccolo da poterlo rappresentare nella notazione semplificata. L'algoritmo si basa sulla divisione Euclidea e ritorna il risultato diviso per ϕ. L'algoritmo è il seguente:

$$
\begin{aligned}
D(\alpha,\ 1:\beta) &= D_1(0:0:\alpha,\ C(\beta)) \\
D_1(0:0:\alpha,\ \beta) &= D_2(A(\alpha,\ 0:\beta,\ 0,\ 0),\ 0:\alpha,\ \beta) \\
D_1(0:1:\alpha,\ \beta) &= D_2(A(\alpha,\ 0:\beta,\ 1,\ 1),\ 1:\alpha,\ \beta) \\
D_1(1:0:\alpha,\ \beta) &= D_2(A(\alpha,\ 1:\beta,\ 1,\ 1),\ \alpha,\ \beta) \\
D_2(0:0:\gamma,\ \alpha,\ \beta) &= 0 : D_1(\alpha,\ \beta) \\
D_2(0:1:0:\gamma,\ \alpha,\ \beta) &= 0 : D_1(\alpha,\ \beta) \\
D_2(0:1:1:\gamma,\ \alpha,\ \beta) &= 1 : D_1(0:0:\gamma,\ \beta) \\
D_2(1:\gamma,\ \alpha,\ \beta) &= 1 : D_1(\gamma,\ \beta)
\end{aligned}
$$

Il cui risultato è:

$$[\![D(\alpha,\ 1:\beta)]\!] = \frac{[\![\alpha]\!]}{[\![\beta]\!] \times \phi}$$

2.8 Divisione completa

Anche l'algoritmo per la divisione completa è tutt'altro che banale: utilizza una funzione ausiliaria, l'algoritmo semplificato, ed entrambi gli algoritmi per il complemento (semplificato e completo). Inoltre è importante sapere che questo algoritmo non termina nel caso in cui si provi a dividere per 0. Le regole sono le seguenti:

$$\begin{array}{lcl}
D'(z:\alpha,\ t:0:\beta) & = & D'_1(C'((z-t):\alpha),\ 0:C(\beta)) \\
D'(z:\alpha,\ t:1:0:\beta) & = & D'(z:\alpha,\ (t-1):\beta) \\
D'(z:\alpha,\ t:1:1:\beta) & = & D'_1((z-t+1):\alpha,\ \beta) \\
D'_1(z:\alpha,\ 0:0:\beta) & = & D'_1((z+1):\alpha,\ \beta) \\
D'_1(z:\alpha,\ 0:1:\beta) & = & (z+1):D_1(A(\alpha,\ \beta,\ 0,\ 0),\ C(\beta)) \\
D'_1(z:\alpha,\ 1:\beta) & = & (z+1):D_1(A(0:\alpha,\ \beta,\ 0,\ 1),\ C(\beta))
\end{array}$$

Capitolo 3

Implementazione

La parte pratica della tesi riguarda principalmente l'implementazione degli algoritmi sopra elencati in Haskell, con l'aggiunta di una funzione che converte numeri razionali di grandezza arbitraria da base 10 a base ϕ e delle due funzioni per i numeri di Fibonacci.

Per convenzione le funzioni che implementano gli algoritmi per la notazione semplificata iniziano con una `s`.

3.1 Il linguaggio Haskell

Il linguaggio scelto per l'implementazione è Haskell, un linguaggio puramente funzionale con delle caratteristiche che si prestano molto allo scopo del calcolo dei numeri reali esatti. Haskell infatti implementa il concetto di lazy evaluation ed è proprio grazie a questo che Haskell permette di utilizzare facilmente gli stream infiniti.

La lazy evaluation, o laziness, è la possibilità di non valutare un'espressione fino a quando non è strettamente necessario. Questo permette di configurare una funzione che genera uno stream infinito senza mai generarlo effettivamente tutto. Se ne genera infatti una cifra alla volta, all'occorrenza. Questo permette di calcolare un numero arbitrario di cifre di un numero reale senza doverlo per forza calcolare tutto, ottenendo così una precisione arbitraria su qualunque computazione di numeri reali.

3.2 Tipi per la rappresentazione

La rappresentazione dei numeri reali in base ϕ tramite notazione semplificata e completa risulta piuttosto immediata quando si parla di Haskell. Per rappresentare uno stream infinito, infatti, basta semplicemente usare una lista di un certo tipo, nel nostro caso una lista di numeri interi:

```
-- un bit dovrebbe essere soltanto 0 e 1 ma,
-- non essendoci un modo immediato per limitare il valore di un intero
-- mantenendo intatto l'aspetto del pattern matching,
-- ho preferito usare direttamente il tipo Int,
-- con la consapevolezza che le uniche cifre valide sono 0 e 1
```

```haskell
type Bit = Int

-- per gli stream in notazione semplificata basta usare una lista di bit
type SNStream = [Bit]

-- per la notazione semplificata ho preferito usare una coppia,
-- in quanto rispecchia maggiormente la convenzione di mantissa-esponente.
-- il tipo Integer rappresenta un intero di grandezza arbitraria:
-- e' necessario per non avere limiti nella rappresentazione dei reali
type FNStream = (Integer, SNStream)
```

3.3 Prodotto completo

Metto ora a confronto gli algoritmi per il prodotto in notazione completa nella versione originale sbagliata e in quella corretta:

```haskell
-- prodotto in notazione completa
-- algoritmo originale sbagliato:
-- restituisce valori completamente fuori scala e talvolta di segno opposto
multiplication' :: FNStream -> FNStream -> FNStream
multiplication' (z, as) (t, bs) = (
        z + t + 2,
        sAddition
            (sMultiplication as bs)
            (sComplement (sAddition as bs 0 0))
            1 0
    )

-- prodotto in notazione completa
-- algoritmo corretto
multiplication :: FNStream -> FNStream -> FNStream
multiplication (z, as) (t, bs) = complement (
        z + t + 2,
        sAddition
            (sComplement (sMultiplication as bs))
            (sAddition as bs 0 0)
            1 0
    )
```

3.4 Costanti

In questa sezione vengono elencate le costanti usate nelle funzioni di conversione. L'algoritmo per ottenere il valore di un numero intero in notazione completa è banale e deriva direttamente dalla definizione di notazione completa:

(1) si rappresenta il numero in base ϕ

(2) si divide il numero per ϕ^2 fino a farlo diventare soltanto decimale, facendolo shiftare a destra di un numero pari di posizioni

(3) si divide ulteriormente il numero per ϕ^2, facendolo shiftare a destra di altre due posizioni

(4) si somma il valore 1 in forma decimale: 0.11

(5) il numero decimale ottenuto è la mantissa del numero in notazione completa

(6) l'esponente è il numero di divisioni fatte durante il secondo passaggio, più una fatta durante il terzo passaggio

Ecco un esempio dell'algoritmo applicato al numero 6:

$$6 \xrightarrow{1} 111.0111 \xrightarrow{2} 1.110111 \xrightarrow{2} 0.01110111 \xrightarrow{3} 0.0001110111 \xrightarrow{4} 0.1101110111 \xrightarrow{5,\ 6}$$

$$6 = \left[\!\left[(3,\ 1:1:0:1:1:1:0:1:1:1:\overrightarrow{0})\right]\!\right]_f$$

```
zeros :: SNStream
zeros = 0:zeros

ones :: SNStream
ones = 1:ones

minusOne :: FNStream
minusOne = (0, zeros)

zero :: FNStream
zero = (0, 1:1:zeros)

one :: FNStream
one = (1, 1:1:1:1:zeros)

phi :: FNStream
phi = (2, 1:1:1:zeros)

two :: FNStream
two = (2, 1:1:0:1:1:1:zeros)

three :: FNStream
three = (2, 1:1:1:1:0:1:zeros)
```

```
four :: FNStream
four = (3, 1:1:0:1:0:1:0:1:zeros)

five :: FNStream
five = (3, 1:1:0:1:0:1:1:1:1:1:zeros)

six :: FNStream
six = (3, 1:1:0:1:1:1:0:1:1:1:zeros)

seven :: FNStream
seven = (3, 1:1:1:0:1:1:0:0:0:1:zeros)

eight :: FNStream
eight = (3, 1:1:1:1:0:1:0:0:0:1:zeros)

nine :: FNStream
nine = (3, 1:1:1:1:0:1:1:1:0:1:zeros)

ten :: FNStream
ten = (3, 1:1:1:1:1:1:0:1:0:1:zeros)
```

3.5 Conversione

Per effettuare la conversione di un numero da base 10 a notazione completa ho scelto un algoritmo semplice: si basa sul valore posizionale delle cifre e le converte una per una, accumulandone le somme dei prodotti per la potenza di 10 corrispondente. Le funzioni sono due:
la prima converte un singolo numero intero da base 10 a notazione completa;
la seconda prende in input due numeri interi, rappresentanti rispettivamente la parte intera e la parte decimale di un numero razionale, e li converte separatamente. Poi divide per 10 la parte decimale di tante volte quante sono necessarie per farle assumere il suo valore decimale inteso nel numero razionale e infine somma le due parti.

Entrambe le funzioni non hanno limite di grandezza sull'input grazie al tipo numerico `Integer`

3.5.1 Conversione di un intero

Per convertire un singolo numero intero da base 10 a notazione completa si converte singolarmente ogni cifra del numero, da destra a sinistra, usando modulo e divisione per 10. Ogni cifra viene moltiplicata per la potenza di 10 che corrisponde alla sua posizione nel numero e si somma alle altre, ottenendo così il risultato finale del numero in notazione completa.

Si sfrutta dunque la seguente proprietà:

$$x_{10} := d_n \dots d_1 \; d_0$$

$$x_{10} = d_n \cdot 10^n + \ldots + d_1 \cdot 10 + d_0$$
$$x_\phi = d_{n\phi} \cdot 10_\phi^n + \ldots + d_{1\phi} \cdot 10_\phi + d_{0\phi}$$

per implementare la seguente funzione:

```
integerToGolden :: Integer -> FNStream
integerToGolden x = if signum x == 1 then real else mul minusOne real where
    integerToGolden' 0 r c = r
    integerToGolden' x r c = case (mod x 10) of
        0 -> integerToGolden' (div x 10)      r                (mul c ten)
        1 -> integerToGolden' (div x 10) (add r (mul one   c)) (mul c ten)
        2 -> integerToGolden' (div x 10) (add r (mul two   c)) (mul c ten)
        3 -> integerToGolden' (div x 10) (add r (mul three c)) (mul c ten)
        4 -> integerToGolden' (div x 10) (add r (mul four  c)) (mul c ten)
        5 -> integerToGolden' (div x 10) (add r (mul five  c)) (mul c ten)
        6 -> integerToGolden' (div x 10) (add r (mul six   c)) (mul c ten)
        7 -> integerToGolden' (div x 10) (add r (mul seven c)) (mul c ten)
        8 -> integerToGolden' (div x 10) (add r (mul eight c)) (mul c ten)
        9 -> integerToGolden' (div x 10) (add r (mul nine  c)) (mul c ten)
    real = integerToGolden' (abs x) zero one
    -- queste righe servono soltanto ad accorciare le righe precedenti
    add = addition
    mul = multiplication
```

In questa funzione il segno viene inizialmente ignorato per convertire il valore assoluto dell'intero in input. Poi viene valutato il segno e si decide se moltiplicare il risultato per -1 o lasciarlo così.

3.5.2 Conversione di un razionale

La conversione di un razionale si basa sulla conversione di due singoli interi: la parte intera e la parte decimale del numero razionale. Queste due parti vengono convertite separatamente e poi sommate, dopo che la parte decimale è stata scalata alla posizione originale dividendola per una potenza di 10.

Si sfrutta dunque la seguente proprietà:

$$x_{10} := d_n \ldots d_1 \; d_0 \; . \; d_{-1} \; d_{-2} \ldots d_{-m}$$

$$x_{10} = d_n \cdot 10^n + \ldots + d_1 \cdot 10 + d_0 + d_{-1} \cdot 10^{-1} + d_{-2} \cdot 10^{-2} + \ldots + d_{-m} \cdot 10^{-m}$$

$$x_{10} = d_n \cdot 10^n + \ldots + d_1 \cdot 10 + d_0 + \frac{d_{-1} \cdot 10^{m-1} + d_{-2} \cdot 10^{m-2} + \ldots + d_{-m}}{10^m}$$

$$x_\phi = d_{n\phi} \cdot 10_\phi^n + \ldots + d_{1\phi} \cdot 10_\phi + d_{0\phi} + \frac{d_{-1\phi} \cdot 10_\phi^{m-1} + d_{-2\phi} \cdot 10_\phi^{m-2} + \ldots + d_{-m\phi}}{10_\phi^m}$$

per implementare la seguente funzione:

```
rationalToGolden :: Integer -> Integer -> FNStream
rationalToGolden i d = if sign then real else mul minusOne real where
    digits = (length . show . abs) d
    integralPart = iToG (abs i)
    decimalPart = division (iToG (abs d)) (iterate (mul ten) one !! digits)
    real = addition integralPart decimalPart
    -- queste righe servono soltanto ad accorciare le righe precedenti
    sign = signum i == signum d
    iToG = integerToGolden
    mul = multiplication
```

In questa funzione il segno dei due interi in input viene inizialmente ignorato per ottenere la conversione dei loro valori assoluti. Poi si procede a controllare se i segni dei due numeri siano concordi o discordi: se sono concordi viene restituito un numero positivo, altrimenti negativo. Questo significa che se in input vengono dati due numeri negativi il risultato sarà comunque positivo.

3.6 Fibonacci

Come detto in precedenza, grazie alla base ϕ è possibile rappresentare e calcolare facilmente i numeri di Fibonacci. Per calcolarli, infatti, esistono due modi particolari che ci permettono di trovare l'n-esimo numero della sequenza senza ricorsione o, in alternativa, con ricorsione ma senza effettuare nessuna operazione matematica sui numeri precedenti. Questa proprietà ci viene dal fatto che possiamo trattare i numeri in base ϕ come delle liste di cifre e, grazie alle proprietà dei numeri di Fibonacci in questa base, possiamo calcolarli facendo soltanto concatenazioni di cifre in testa ad una lista.

Va notato, però, che per semplicità le due funzioni non restituiscono un numero in notazione completa bensì una coppia $(e,\ m)$ dove e sta per esponente e m per mantissa: il significato di questa coppia è esattamente quello della notazione esponente-mantissa e una grande differenza rispetto alla notazione completa è che la mantissa in questo caso non è infinita ma contiene soltanto il minimo numero di bit necessari a rappresentare il numero. Volendo trasformare questo risultato in notazione completa basterebbe seguire l'algoritmo nella sezione 3.4.

3.6.1 Senza ricorsione

La funzione per calcolare i numeri di Fibonacci senza basarsi sui numeri precedenti (ovvero senza ricorsione) è la seguente:

```
fibNoRec :: Integer -> (Integer, [Bit])
fibNoRec n = (n - 1, fs) where
    digits 0 s = s
    digits x s = 1:0:0:0:digits (x - 1) s
    -- queste righe servono soltanto ad accorciare le righe precedenti
    fs = digits (div (n - 1) 2) (if mod n 2 == 0 then [1] else [1, 1])
```

Si basa sul fatto che il numero $Fib(n)$ ha $n-1$ bit nella parte intera e i restanti nella parte decimale, ed è sempre composto dalle cifre 1000 concatenate per $\frac{n-1}{2}$ volte, come si può notare dalla tabella nella sezione 1.2.4.

Per precisione va detto che in questa funzione la ricorsione viene usata non sulla funzione `fibNoRec` ma sulla funzione `digits`, necessaria soltanto per generare la mantissa.

3.6.2 Con ricorsione

La funzione per calcolare i numeri di Fibonacci con ricorsione ma senza nessuna operazione matematica sui risultati precedenti è la seguente:

```
fibRec :: Integer -> (Integer, [Bit])
fibRec 1 = (0, [1, 1])
fibRec 2 = (1, [1])
fibRec n = (f + 2, 1:0:0:0:fs) where
    (f, fs) = fibRec (n - 2)
```

Si basa sul fatto che il numero $Fib(n)$ può essere ottenuto concatenando le cifre 1000 in testa al numero $Fib(n-2)$ e incrementando di due l'esponente, come si può notare dalla tabella nella sezione 1.2.4 [**?**].

Bibliografia

[1] di Gianantonio, P. A golden ratio notation for the real numbers.

[2] Henney, K. *97 Things Every Programmer Should Know: Collective Wisdom from the Experts.* O'Reilly Media, 1 edn. (2010). ISBN 9780596809485,0596809484.

[3] Plume, D. A calculator for exact real number computation. (1998).