

## Ett komplementär läromaterial för datastudenter som lär sig fysik

Läromaterialet Learn You a Physics for Great Good!

Kandidatarbete inom Data- och informationsteknik

Johan Johansson

Oskar Lundström

Erik Sjöström

Björn Werner



KANDIDATARBETE DATX02-18-08

# Ett komplementerande läromaterial för datastudenter som lär sig fysik

Läromaterialet Learn You a Physics for Great Good!

Johan Johansson  
Oskar Lundström  
Erik Sjöström  
Björn Werner



**CHALMERS**

Institutionen för Data- och informationsteknik  
*Avdelningen för funktionell programmering*  
CHALMERS TEKNISKA HÖGSKOLA  
Göteborg, Sverige 2018

Ett komplementerande läromaterial för datastudenter som lär sig fysik  
Läromaterialet Learn You a Physics for Great Good!

Johan Johansson  
Oskar Lundström  
Erik Sjöström  
Björn Werner

© Johan Johansson, 2018.  
© Oskar Lundström, 2018.  
© Erik Sjöström, 2018.  
© Björn Werner, 2018.

Handledare: Patrik Jansson, CSE, Chalmers

Kandidatarbete DATX02-18-08  
Institutionen för Data- och informationsteknik  
Avdelningen för funktionell programmering  
Chalmers tekniska högskola  
SE-412 96 Göteborg  
Telefon +46 31 772 1000

Omslag: Newton står och tänker på fritt fall i termer av Haskellkod och precis då får han ett lambda-äpple i huvudet. Bilden är ritad i samma stil som bilderna i läromaterialet.

Typsatt i L<sup>A</sup>T<sub>E</sub>X  
Göteborg, Sverige 2018

# Abstract

This report describes the development of the learning material *Learn You a Physics for Great Good!*. The material was developed as part of a bachelor thesis project at the Department of Computer Science and Engineering at Chalmers University of Technology. The goal of the project is to create a text-based learning material which presents physics with the help of the programming concept *domain specific languages*, where the domain specific languages are implemented in the programming language Haskell. Furthermore, the pedagogical use of a learning material of this kind and how well physics and domain specific languages can be combined are discussed.

The background of the project is the (for Computer Science and Engineering students at Chalmers) mandatory physics course *Fysik för ingenjörer* which has had less-than-good exam statistics for several years. We believe one factor for why Computer Science and Engineering students get bad results in this course is that they find physics irrelevant in relation to the rest of their education. We think this problem can be solved with a learning material bridging physics and programming, both highlighting the relevancy of physics and invoking an interest for it. An increased interest will hopefully lead to better results in the course.

The learning material includes five chapters dealing with physical dimensions, calculus, vectors, particle mechanics and applications of them. Each chapter consists of Haskell code combined with a descriptive text. Some chapters construct domain specific languages from the ground up, while some apply previously constructed domain specific languages on physical problems. The learning material is published on a website<sup>1</sup> and its source code is freely available<sup>2</sup>.

The report also describes the meetings and discussions conducted with non-project members. Their purpose has been to improve the learning material and to perform an informal evaluation of the completed learning material.

The conclusions are that domain specific languages can have a pedagogical use in physics education. The rigorous nature of domain specific languages makes the process of physical problem solving rigorous as well. We believe this way of thinking can further the teaching if it's used in traditional physics education.

Keywords: Domain Specific Languages, Classical Mechanics, Physics Education, Learning Material, Functional Programming, Haskell

---

<sup>1</sup><https://dslsofmath.github.io/BScProj2018/>

<sup>2</sup><https://github.com/DSLsofMath/BScProj2018>



# Sammanfattning

Denna rapport beskriver utvecklingen av läromaterialet *Learn You a Physics for Great Good!*. Materialet utvecklades som ett kandidatarbete vid institutionen för Data- och informationsteknik på Chalmers tekniska högskola. Syftet med projektet är att skapa ett textbaserat läromaterial som presenterar fysik med hjälp av programmeringskonceptet *domänspecifika språk*, där de domänspecifika språken är implementerade i programmeringsspråket Haskell. Vidare diskuteras den pedagogiska nyttan av ett läromaterial av detta slag samt hur väl fysik och domänspecifika språk går att kombinera.

Bakgrunden till projektet är fysikkursen *Fysik för ingenjörer*. Kursen är obligatorisk för studenter på Datateknik (D) på Chalmers och har haft ganska dålig tentastatistik i flera år. Vi tror att en faktor till att just D-studenter får dåliga resultat i denna kurs är att studenterna finner ämnet irrelevant i förhållande till resten av utbildningen. Vi tror också att detta problem kan lösas med ett läromaterial som brygger fysik och programmering, och både visar på relevansen av ämnet och väcker intresse för fysik. Ett ökat intresse för fysik leder förhoppningsvis till bättre resultat i kursen.

Det resulterande läromaterialet innehåller fem kapitel som behandlar områdena fysikaliska dimensioner, matematisk analys, vektorer och partikelmekanik, och tillämpningar av dem. Varje kapitel består av Haskell-kod tillsammans med beskrivande text. Vissa kapitel bygger upp domänspecifika språk från grunden medan andra kombinerar och tillämpar tidigare domänspecifika språk på fysikaliska problem. Läromaterialet är publicerat på en hemsida<sup>3</sup> och dess källkod finns fritt tillgänglig<sup>4</sup>.

Rapporten beskriver även de möten och diskussioner som genomförts med utomstående. Syftet var att förbättra läromaterialet samt att genomföra en informellt test av läromaterialet.

Slutsatserna är att domänspecifika språk kan ha en pedagogisk nytta i fysikundervisning. Den rigorösa naturen hos Haskell och domänspecifika språk gör att den fysikproblemlösningen de används för också blir rigorös, utan möjlighet till intuitiva men felaktiga genvägar. Detta tankesätt tror vi kan främja lärande om det integreras i traditionell fysikundervisning.

Nyckelord: Domänspecifika språk, Klassisk mekanik, Fysikundervisning, Läromaterial, Funktionell programmering, Haskell

<sup>3</sup><https://dslsofmath.github.io/BScProj2018/>

<sup>4</sup><https://github.com/DSLsofMath/BScProj2018>





# Förord

Denna rapport behandlar kandidatarbetet *Matematikens domänspecifika språk*, som genomfördes på Chalmers tekniska högskola under vårterminen 2018. Vi som har utfört detta kandidatarbete är tre studenter från civilingenjörsprogrammet Datateknik vid Chalmers tekniska högskola och en student från det datavetenskapliga programmet vid Göteborgs universitet.

Vi vill tacka Patrik Jansson, vår handledare, som med sina kloka tankar och goda råd agerat som ett fyrtorn när vi seglat på okända domänspecifika hav. Vi vill tacka Åke Fäldt som tagit sig tiden att diskutera sin egen kurs, vårt läromaterial och gett oss tips och råd under utvecklingen. Vi vill tacka de testare, både individer och grupper, som tagit sig tiden att studera och läsa igenom vårt, ibland halvfärdiga, material och gett oss den kritik vi behövt för att spurras till vidareutveckling. Vi vill tacka Jeff Chen vars tankar och idéer om potentiella vidareutvecklingar av projektet gav oss ett helt nytt perspektiv under arbetets gång.

Slutligen vill vi tacka Miran Lipovača vars hemsida “Learn You a Haskell for Great Good!” har både inspirerat utformningen av vår hemsida och agerat som ett läromaterial för våra egna inledande studier av det fantastiska programmeringsspråket Haskell.

Författarna, Göteborg, maj 2018.



# Innehåll

<b>Ordlista</b>	<b>xiii</b>
<b>1 Introduktion</b>	<b>1</b>
1.1 Bakgrund . . . . .	1
1.2 Projektets mål . . . . .	2
1.3 Avgränsningar . . . . .	3
<b>2 Teori</b>	<b>5</b>
2.1 Domänspecifika språk . . . . .	5
2.2 Syntax och semantik . . . . .	6
2.3 Litterat programmering och Literate Haskell . . . . .	7
2.4 Lärandeteorier . . . . .	8
<b>3 Genomförande</b>	<b>11</b>
3.1 Konstruktion av läromaterialet . . . . .	11
3.1.1 Sökande efter områden att behandla . . . . .	11
3.1.2 Implementation av domänspecifika språk för områdena . . . . .	12
3.1.3 Skriva lärottext . . . . .	12
3.2 Möten och återkoppling . . . . .	13
<b>4 Resultat</b>	<b>15</b>
4.1 Läromaterialet . . . . .	15
4.1.1 Översikt . . . . .	15
4.1.2 Dimensioner . . . . .	17
4.1.3 Matematisk analys . . . . .	18
4.1.4 Vektorer . . . . .	19
4.1.5 Exempelproblem . . . . .	20
4.1.6 Partikelmekanik . . . . .	21
4.2 Möten och återkoppling . . . . .	21
<b>5 Diskussion</b>	<b>23</b>
5.1 Genomförandediskussion . . . . .	23
5.2 Resultatdiskussion . . . . .	24
5.2.1 Om läromaterialets fokus på matematik och Haskell snarare än fysik	25
5.2.2 Lämpliga områden för domänspecifika språk . . . . .	26
5.2.3 Domänspecifika språk, fysik och pedagogiska aspekter . . . . .	27
5.3 Vidareutvecklingsmöjligheter och behov av ytterligare kunskap . . . . .	28
5.4 Etiska aspekter . . . . .	29

<b>6</b>	<b>Slutsatser</b>	<b>31</b>
	<b>Bibliography</b>	<b>33</b>
<b>A</b>	<b>Läromaterialet</b>	<b>I</b>

# Ordlista

**Datastudent** En student med bakgrund inom data och IT.

**Dimensioner** Syftar på *fysikaliska* dimensioner.

**DSL** Förkortning av *Domain Specific Language*, engelska för domänspecifikt språk.

**DSLsofMath** Förkortning av *Domain Specific Languages of Mathematics*, vilket är en valbar kurs i årskurs 2 på Chalmers tekniska högskola.

**Fysik för ingenjörer** En fysikkurs som är obligatorisk för studenter på civilingenjörsprogrammet Datateknik på Chalmers. Den ges i årskurs 2 och innehåller grunderna i klassisk mekanik, termodynamik och vågrörelselära.

**Literate Haskell** Litterat programmering i Haskell. Se avsnitt 2.3

**Läromaterialet** Syftar på den pedagogiska text som projektet resulterat i.

**Semantik** Vad meningar (skapade i en syntax) har för betydelse. Se avsnitt 2.2.

**Syntax** Grammatiken för ett språk, som beskriver hur meningar konstrueras i det. Se avsnitt 2.2.

**Åke Fäldt** Föreläsare och examinator i kursen Fysik för ingenjörer.



# 1

## Introduktion

Detta kapitel beskriver projektets bakgrund, mål och avgränsningar.

### 1.1 Bakgrund

På civilingenjörsprogrammet Datateknik på Chalmers tekniska högskola ingår den obligatoriska fysikkursen *Fysik för ingenjörer*. Tentastatistiken för denna kurs [1] är ganska dålig<sup>5</sup>. Vi tror att många studenter på Datateknik tycker att denna kurs är svår eller ointressant och att detta leder till att en betydande andel får underkänt.

Detta tror vi kan lösas med avstamp från kursen *Domain Specific Languages of Mathematics* (“DSLsofMath”), med den svenska titeln *Matematikens domänspecifika språk*. Kursen är valbar på kandidatnivå för studenter på Chalmers och Göteborgs universitet. Konkret presenterar DSLsofMath matematik som derivator, komplexa tal och matriser ur ett programmeringsperspektiv i Haskell, vilket är ett programmeringsspråk datastudenterna redan är bekanta med.

DSLsofMath-kursens skapare, Cezar Ionescu och Patrik Jansson, har beskrivit avsikten med kursen i en artikel [2]. Det direkta målet med kursen är att förbättra den matematiska utbildningen för datavetare och den datavetenskapliga utbildningen för matematiker, där den grundläggande idén bakom kursen är:

“[...] att uppmuntra studenterna att närma sig matematiska domäner från ett funktionellt programmeringsperspektiv: att ge beräkningsbevis (calculational proofs); att vara uppmärksamma på syntaxen för matematiska uttryck; och, slutligen, att organisera de resulterande funktionerna och typerna i domänspecifika språk.” [2]

Det programmeringsperspektiv som kursen använder sig av botten i så kallade domänspecifika språk. Kortfattat kan ett domänspecifikt språk beskrivas som ett programmeringsspråk som skapats för ett väl avgränsat område. Detta område kan vara databashantering, algebraiska uttryck eller till och med fysik. Språket kan antingen vara implementerat inuti ett annat programmeringsspråk eller implementerat helt fristående. I kursen och projektet är de implementerade i Haskell.

---

<sup>5</sup>Andel underkända på ordinarie tentamen från läsår 2014 till 2017: 34%, 76%, 18%, 57%.

Idén bakom projektet är att använda domänspecifika språk för att ur ett alternativt perspektiv presentera fysik. Likt det sätt DSLsofMath presenterar kopplingar mellan matematik och programmering ska projektet på motsvarande sätt visa på kopplingar mellan programmering och fysik och därmed underlätta lärandet. För att förtydliga ges här en analogi:

Studenterna har svårt för matematik  $\rightarrow$  DSLsofMath.  
Studenterna har svårt för fysik  $\rightarrow$  Detta projekt.

Detta projekt kan vara av intresse för studenter, pedagoger och föreläsare inom de berörda områdena eftersom projektet ger ett nytt perspektiv på fysik som inte bara är annorlunda utan också mer rigoröst. Förhoppningsvis blir det även relevant för de som är intresserade av domänspecifika språk i stort och kanske till och med för programledningen som kan se denna rapport som ett skäl att introducera innehåll av detta slag i fysikkurser.

Angående tidigare forskning och studier har en kurs på MIT, inte helt olik DSLsofMath, tidigare givits som berör både fysik och domänspecifika språk. *Classical Mechanics: A Computational Approach* gavs av Gerald Sussman och Jack Wisdom senast år 2008 [3]. Denna kurs på avancerad nivå behandlar de fundamentala principerna för klassisk mekanik med hjälp av beräkningsidéer för att precis formulera principerna av mekanik, med början i Lagranges ekvationer och avslut i perturbationsteori (teori för approximationer av matematiska lösningar). I kursens bok [4] förklaras fysikaliska fenomen genom att visa datorprogram för att simulera dem, skrivna i språket Scheme. Denna typ av kurs är ovanlig och är, till projektgruppens kännedom, den enda kursen bortsett från DSLsofMath som knyter samman fysik, programmering och matematik på en symbolisk nivå för att förklara koncepten. Skillnaden mot vårt projekt är att vi använder Haskell istället för Scheme, enbart behandlar fysik från Fysik för ingenjörer samt gör det på en nivå anpassad för datastudenter på Chalmers.

Även tidigare har det genomförts ett kandidatarbete på Chalmers med anknytning till DSLsofMath. Vårterminen 2016 genomfördes kandidatarbetet *Programmering som undervisningsverktyg för Transformer, signaler och system. Utvecklingen av läromaterialet TSS med DSL* av fem studenter från Datateknik och Teknisk Matematik på Chalmers [5]. Arbetet bestod av utveckling av läromaterial med tillhörande programmeringskod, uppgifter och lösningar, som komplement till existerande kurser i signallära.

Till sist finns det ett arbete som liknar detta arbete i både syfte och programmeringsspråk, vilket utfördes av Scott N. Walck vid Lebanon Valley College [6]. Syftet med det projektet var att fördjupa studenters förståelse av fysik, med fokus på elektromagnetisk teori, genom att uttrycka fysiken med hjälp av funktionell programmering. Skillnaden är att detta projekt ska förklara fysik från grunden.

## 1.2 Projektets mål

Målet med detta kandidatarbete är att angripa fysik från ett programmeringsperspektiv, förhoppningen är då att fysik ska bli både roligare och intressantare för datastudenter,



och därmed också enklare. Detta liknar premissen bakom kursen DSLsofMath och kandidatarbetet från 2016, som istället för fysik behandlade matematik respektive signallära.

Mer konkret ska det ovanstående målet uppnås genom att skapa ett läromaterial. Läromaterialet ska bestå av domänspecifika språk som modellerar fysik, skrivna i Haskell, sammanvävt med en förklarande lärottext. Läromaterialet ska vara enkelt för läsaren att ta till sig, vilket ska åstadkommas genom ett lättsamt språk, publicering på en hemsida samt fri tillgång till källkoden.

Ett parallellt mål är att, efter att ha tillägnat sig erfarenhet, diskutera huruvida fysik och domänspecifika språk går att kombinera och om det finns en pedagogisk nytta i att göra det.

### 1.3 Avgränsningar

Läromaterialet begränsas till att enbart hantera de fysikaliska områden som ingår i kursen Fysik för ingenjörer. Denna avgränsning valdes dels eftersom det är den fysik gruppmedlemmarnas kunskaper är begränsad till, och dels för att det är denna kurs som projektet kan bli mest relevant för, eftersom kursen ingår i datastudenternas obligatoriska kursplan.

Vidare ska en prioritering av innehållet i Fysik för ingenjörer göras. Kursen behandlar grunderna inom klassisk mekanik, termodynamik och vågrörelselära samt en stor mängd tillämpad matematik, exempelvis differentialkalkyl. I första hand behandlas mekaniken, för att sedan i mån av tid även behandla termodynamik och vågrörelselära eftersom det är i den ordningen kursen behandlar områdena. Fokuset läggs även på de områden datastudenter haft svårt för.

För att pröva den pedagogiska nyttan kommer enbart en informell testning att göras. Detta då en rigorös undersökning hade krävt mycket tid för att välja lämpliga testgrupper, analysera återkopplingen samt dokumentera testningsförloppet. Denna tid läggs istället på att skapa ett intressant innehåll.

Projektet fokuserar mer på att skapa innehåll än att göra efterforskningar på, och tillämpa, pedagogiska teorier och riktlinjer. Denna avgränsning valdes eftersom det är hur innehållet kan se ut som är intressant och nytt, inte hur ett pedagogiskt läromaterial kan skrivas på bästa sätt. Den pedagogiska aspekten kommer inte ignoreras helt, fokuset på den kommer bara att vara mindre.



# 2

## Teori

I detta kapitel beskrivs fyra koncept av central betydelse för projektet. Dessa koncept är domänspecifika språk, begreppen syntax och semantik, litterat programmering samt lärandeteorier.

### 2.1 Domänspecifika språk

Ett domänspecifikt språk är ett språk som är avgränsat till en specifik domän. Nyckelorden är språk, specifik och domän. En domän är ett område, till exempel textformatering eller matlagning. Specifikt syftar det på att det är *just detta* område som fokus läggs på. Med språk menas ett sätt att uttrycka saker inom domänen. Svenska och Java är två exempel på språk. Domänspecifika språk är vanligt förekommande i programmeringssammanhang, HTML är ett domänspecifikt språk för textformatering, SQL för databashantering och CSV för tabeller.

Domänspecifika språk används inte bara i programmering utan förekommer även i andra mer vardagliga sammanhang. Inom domänen matlagning är steka, grilla och fritera användbara ord, likaså inom domänen ridning är gramma, box och galopp användbara ord. Är personen bekant med domänen vet den vad som menas med gramma och det är ett kort och väldefinierat sätt att uttrycka sig. Men detta språk (här i form av ord och begrepp) blir svårtolkat utanför domänen. Ett recept kan inte förklaras i termer av grimmor, boxar och galopper. Precis som domänspecifika språk i vardagen passar domänspecifika språk inom programmering bäst för sin egen domän. SQL är bra för att hantera en databas men inte för att skapa ett spel.

Motsatsen till ett domänspecifikt språk är ett generellt språk. I vardagen är naturliga språk som svenska och engelska generella medan ryttarbegreppen ovan är domänspecifika. Precis som i vardagen finns det i datavärlden generella programmeringsspråk, till exempel C++ och Java. Dessa språk har inte en specifik domän utan kan användas till flera olika, Java kan till exempel användas till både databaser och spel. Nackdelen med dessa generella språk är just att de är så generella. Eftersom domänspecifika språk inte behöver vara användbara utanför den specifika domänen kan de inkludera specifik syntax och ha en inbyggd funktionalitet som inte hade passat i ett generellt språk.

Ett domänspecifikt språk kan antingen implementeras som ett fristående språk eller bäddas in i ett redan existerande språk. De domänspecifika språk som utvecklats inom detta projekt är inbäddade i programmeringsspråket *Haskell*. Haskell är ett lämpligt val eftersom det är enkelt att skapa datatyper som bygger upp det domänspecifika språket. Att Haskell är ett högnivåspråk är också en fördel då programmeraren slipper programmeringstekniska detaljer som minneshantering och kan istället fokusera på programmets innehåll och betydelse. Slutligen möjliggör mönstermatchning att de konstruktörer som bygger upp datatyperna i det domänspecifika språket enkelt kan brytas isär och manipuleras.

För vidare läsning om domänspecifika språk rekommenderas *DSL for the Uninitiated* [7].

## 2.2 Syntax och semantik

I samband med domänspecifika språk dyker begreppen *syntax* och *semantik* upp. Syntax är reglerna för hur enheter i språket, till exempel ord och skiljetecken, sammanslås till komplexa strukturer som meningar och satser. Semantiken är betydelsen av sådana komplexa strukturer i ett språk. Inom aritmetik<sup>6</sup> är tal och operationer syntax medan värdet av uttrycket är semantiken. Till exempel har det syntaktiska uttrycket  $7 * (3 + 10)$  det semantiska värdet 91, eftersom det är det som det syntaktiska uttrycket *betyder*. Domänspecifika språk har med syntax att göra eftersom många domänspecifika språk används för att modellera just syntax.

För att illustrera visas här ett domänspecifikt språk i Haskell som modellerar en del av syntaxen för aritmetiska uttryck.

```
data Expr = Expr :+: Expr
          | Expr **: Expr
          | Const Double
```

Typen innehåller *datakonstruktörer* för att representera *ändpunkter* och *förgreningar*. I detta exempel är `:+:` och `**:` förgreningar. Med hjälp av dem kan summan respektive produkten av två andra uttryck representeras. Ändpunkterna representeras av `Const`.

Med hjälp av datakonstruktörerna ovan kan aritmetiska uttryck konstrueras. Till exempel kan  $7 * (3 + 10)$  modelleras som

```
expr = Const 7 **: (Const 3 :+: Const 10)
```

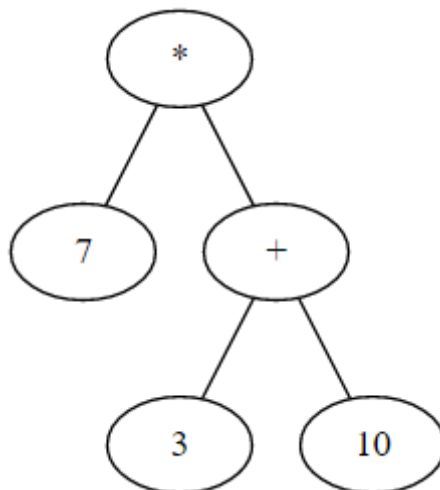
Konstruktorn `**:` får som sina två argument uttrycken `Const 7` och `Const 3 :+: Const 10`. Det är alltså en produkt av två deluttryck.

Ett syntaktiskt uttryck kan visualiseras med ett trädigram, ibland kallat *syntaxträd*. Exempeluttrycket ovan illustreras i figur 2.1.

För att beräkna semantiken hos ett syntaktiskt uttryck representerat i Haskell används en

---

<sup>6</sup>Aritmetik är den gren inom matematiken som behandlar räkning av tal.



**Figur 2.1:** Ett exempeluttryck från syntaxen `Expr` visualiserat i ett trädidiagram.

*beräkningsfunktion.* I fallet med aritmetiska uttryck är semantiken ett numeriskt värde. För `Expr` kan beräkningsfunktionen se ut som

```

evaluate :: Expr -> Double
evaluate (e1 :+: e2) = evaluate e1 + evaluate e2
evaluate (e1 :+: e2) = evaluate e1 * evaluate e2
evaluate (Const v)  = v
  
```

Det finns tre speciella saker att observera. Den första är att eftersom syntaxen innehåller tre olika sorters element, här motsvarat av de tre datakonstruktorerna, används tre fall i funktionen `evaluate` som beräknar vardera av dem. Den har därför ett fall för `:+:`, ett för `:+:` och ett för `Const`.

Den andra saken att notera är hur ett fall implementeras. Här är `e1 :+: e2` syntax för addition av de två uttrycken `e1` och `e2`. Semantiken av detta är inom aritmetik definierad som summan av värdena hos de två deluttrycken. Implementationen av detta fall ska därför vara `evaluate e1 + evaluate e2`. Ett liknande resonemang ger svaret på hur beräkningen av de två resterande fallen ska se ut.

Den tredje saken värd att poängtera är beräkningsfunktionens typsignatur, `Expr -> Double`. Den gör nämligen att `evaluate`, och beräkningsfunktioner i allmänhet, kan tolkas som en översättning från syntax (här `Expr`) till semantik (här `Double`).

## 2.3 Litterat programmering och Literate Haskell

*Litterat programmering* (engelska *literate programming*) är ett alternativt sätt att programmera som introducerades av Donald Knuth [8]. Istället för att skriva ett program främst för datorer att exekvera, så skrivs programmet främst för människor att läsa.

Jämfört med traditionella program får dokumentationen en ökad betydelse. I traditionella

program är programkoden den viktiga delen. I litterata program är däremot dokumentationen minst lika viktig. Den används för att förklara koden, sätta den i relation till andra delar, med mera. Detta jämbördiga förhållande syns konkret genom att titta på hur källkoden är skriven i ett litterat program. Det kan till exempel se ut som i figur 2.2 där källkoden och dokumentationen är sammanvävda på ett jämbördigt sätt, där den ena inte är viktigare än den andra.

```
How does all this tie together? First the type is decided ,
  for instance

> type ExampleType = Quantity T.Length Double

then a value of that type is created

> exampleValue :: ExampleType
> exampleValue = Quantity V.length 5.3

Note that the Quantity data type has both value-level and
  type-level dimensions. As previously mentioned, value-
  level in order to pretty print and type-level to only
  permit legal operations.
```

**Figur 2.2:** Ett exempel på hur en källfil till litterat programmering kan se ut, tagen direkt från källkoden till läromaterialet. I exemplet är koden skriven i Literate Haskell. Rader som börjar med > markerar att det är programkod, medan rader utan markerar att det är dokumentation.

*Literate Haskell* är litterat programmering för Haskell [9]. Att programmera i Literate Haskell går till på samma sätt som vanlig Haskell, med skillnaden att programkod och text vävs ihop i en och samma fil. Det kan se ut som i figur 2.2. Filen, med tillägget `.lhs`, går att använda direkt med Haskell-kompilatorn GHC. All text ignoreras och programkoden behandlas som om den var en vanlig Haskell-fil. Filen kan också kompileras till en läsbar typsatt rapport eller hemsida. Det som används i detta projekt är *Pandoc* [10]. Med Pandoc kan texten märkas upp med både *Markdown* (används i projektet) och  $\LaTeX$ . Det går att exportera till bland annat HTML (som i läromaterialet) och PDF (som i den här rapporten).

## 2.4 Lärandeteorier

I detta projekt är lärandeteorier om motivation, lärande i grupp, interaktion och snabba belöningar relevanta.

Motivation är en persons vilja att göra något och i undervisningssammanhang vill läraren eller författaren att studenten ska lära sig materialet, studenten behöver alltså vara motiverad för att lära sig. Motivation kan ha flera källor, till exempel att studenten tycker materialet är intressant eller att det finns belöningar i form av tillfredsställelsen från ett högt betyg.

*Motiverande design* innebär att systematiskt utforma undervisningen på ett sådant sätt att studenten blir motiverad till att lära sig. Det handlar om att använda olika tekniker för att väcka och behålla motivation. För detta finns det ett flertal olika modeller men i detta projekt används enbart den så kallade *ARCS-modellen* [11].

*ARCS* är en förkortning av “Attention, Relevance, Confidence and Satisfaction”, på svenska “uppmärksamhet, relevans, självförtroende och tillfredsställelse”. Precis som namnet antyder innehåller modellen fyra delar som vardera behandlar en aspekt av motivation.

- *Attention* handlar om att fånga uppmärksamhet och väcka nyfikenhet.
- *Relevance* handlar om att tillgodose studentens behov så att materialet upplevs som relevant.
- *Confidence* handlar om att övertyga studenten att hen kan lyckas lära sig materialet.
- *Satisfaction* handlar om att ge studenten tillfredsställelse efter att ha lärt sig något så att hen vill fortsätta lära sig.

Det finns olika strategier för hur de olika delarna genomförs i praktiken, här följer en översikt för *Attention*<sup>7</sup>.

För att fånga studentens uppmärksamhet och intresse finns tre allmänna strategier. Den första är varseblivning, att något plötsligt händer som studenten blir medveten om. Det kan till exempel åstadkommas genom överraskande information, en förändring i ljuset i en föreläsning eller att humor vävs in. Den andra är att väcka nyfikenhet. Ett par sätt för det är att involvera mystik i miljön och att ställa frågor. Det tredje sättet är variation. Då handlar det om olika struktur och ordning på undervisningen, till exempel att inte alltid utforma en lektion som föreläsning, demonstration och sedan övning, utan variera det med andra inslag, exempelvis ett filmklipp.

Lärandeteorier om hur elever lär sig i grupp är också relevanta. Utifrån det sociokulturella perspektivet som Vygotskij utvecklade [12] lär sig elever av varandra. Eleverna befinner sig i sin *närmsta utvecklingszon*<sup>8</sup>, där eleverna kan hjälpa varandra att förstå innebörden av definitioner och uttryck genom att sätta ord på det de vill kommunicera. Denna typ av kommunikation kan hjälpa elever sätta fingret på vad de inte förstår. Med denna bakgrund kan parprogrammering vara fördelaktigt. Dels för att eleverna kan lära sig av varandra, dels för att de genom att kommunicera internaliserar ämnet och bygger en djupare förståelse. Parprogrammering kan även lämpa sig för att begränsa flyktt försök, där elever medvetet eller mindre medvetet börjar ägna sig åt något annat.

Två andra aspekter på lärande är interaktion och snabba belöningar. Eftersom internetplattformars interaktion med eleven är begränsad i jämförelse med då eleven är i skolan och har tillgång till lärare, så brukar internetbaserade läroplattformar förlita sig på behavioristiska element (former av respons) i form av rätt eller fel svar [12]. Evolutionärt sett har snabba belöningar varit fördelaktigt framför långsiktiga som kräver långsiktigt engagemang (exempelvis öva inför en tenta) vilket beskrivs i boken *Dansa på deadline*:

<sup>7</sup>Eftersom projektet har ett begränsat fokus på de pedagogiska aspekterna, se avsnitt 1.3, har enbart *Attention* tagits hänsyn till. Av detta skäl är enbart denna del beskriven här.

<sup>8</sup>Den zon där målet för lärandet ligger på en nivå som är för hög för en elev att klara på egen hand, men som eleven klarar om den får vägledning och stöd.

*Uppskjutandets psykologi* [13]. Dessa två aspekter återkommer i kapitel 5



# 3

## Genomförande

Projektets genomförande bestod av fyra delar. Den största delen var konstruktionen av själva läromaterialet i vilken det ingick sökande efter lämpliga områden, implementation av domänspecifika språk och skrivande av lärotext. De tre andra delarna var publicering av läromaterialet på en hemsida, återkoppling på läromaterialet med en testgrupp samt möten med Åke Fäldt, examinator och föreläsare för Fysik för ingenjörer. Mötena med Fäldt hade två syften: att hitta problemområden i fysikkursen och att få återkoppling på läromaterialet. Alla de olika delar i projektet genomfördes samtidigt men de finns här beskrivna separat.

### 3.1 Konstruktion av läromaterialet

Läromaterialet består av fem kapitel som vardera behandlar separata områden. Skapandet av dem skedde fristående men de innehöll alla de tre faserna sökande, implementation och skrivande, som såg likartade ut för dem alla. Det fanns dock visst överlapp mellan de fristående processerna. Sökandet gav ofta flera områden samtidigt och implementation och skrivande genomfördes ofta parallellt. För att tydliggöra processerna är de dock beskrivna separat.

#### 3.1.1 Sökande efter områden att behandla

Ett domänspecifikt språk modellerar ett specifikt och avgränsat område. Därför var det naturligt att söka och tänka i termer av avgränsade områden inom fysiken. För att rent praktiskt hitta områden att behandla kontaktades Åke Fäldt. Dessutom studerades kursens bok (University Physics [14]) och dess övriga material. Av speciellt intresse var de kapitel som behandlade mekanik (i enlighet med projektets mål att börja med klassisk mekanik), de områden Fäldt pekade ut som svåra för studenter samt de kapitel som använde sig av en specifik syntax. Domänspecifik syntax var av intresse att finna då en betydlig del av domänspecifika språk är modellering av just syntaxen. Denna sökandeprocess innefattade inte bara att *hitta* fysikaliska områden utan även att *organisera* dem i relation till varandra, för att kunna implementera dem på ett bra sätt och presentera dem i en pedagogisk ordning.

#### 3.1.2 Implementation av domänspecifika språk för områdena

Implementationen av domänspecifika språk var en iterativ process. Det finns inte bara ett rätt sätt att skriva ett domänspecifikt språk på, därav gjordes försök med flera olika varianter för att se vad som fungerade bäst. I flera fall har implementationer gjorts om från grunden om det visat sig att implementationen kunde gjorts bättre eller hade brister. Dessutom gjordes, i varierande mån, fördjupande litteraturstudier av domänspecifika språk, fysik och Haskell för att kunna implementera de domänspecifika språken på bästa sätt.

Vad som ansågs vara en bra, eller åtminstone tillräckligt bra implementation var i huvudsak baserat på gruppmedlemmarnas erfarenhet av Haskell och diskussion inom gruppen och med handledaren, det viktigaste var att de var lättförståeliga. Därför användes inte alltid den programtekniskt elegantaste implementationen utan den längre versionen föredrogs för att göra läromaterialet så lättläst som möjligt. Dock avstods det inte från användning av mer avancerade funktioner i Haskell när materialet motiverade dem, men då alltid med en uttömmande förklaring om hur det fungerade och utan krav på tidigare kunskap hos läsaren.

Efter att ett domänspecifikt språk implementerats skrevs tester till det. Det som var intressant att testa var huruvida domänens lagar gällde i det domänspecifika språket som modellerade domänen. Till exempel var det för det domänspecifika språket om vektorer aktuellt att testa om vektoraddition var kommutativ, och så vidare. Testerna gjordes med hjälp av *QuickCheck* [15] vilket är ett testningsverktyg i Haskell som genererar många och slumpmässiga testfall. Att lagarna gällde för de domänspecifika språken verifierades med andra ord genom testa för ett stort antal exempelvärden. Inga bevis, utan enbart tester, gjordes för att kontrollera att lagarna gällde.

#### 3.1.3 Skriva lärotext

Generellt under skrivningen togs det hänsyn till en specifik underaspekt i ARCS-modellen, nämligen *humor*. Språket i lärotexten har varit lättsamt, vardagligt och talspråkligt för att hålla kvar uppmärksamheten hos läsaren. Det har även ritats roliga bilder för att ge ytterligare humoristiska drag.

Lärotexten och programkoden skrevs sammanvävt i samma fil, i *Literate Haskell*. Litterat programmering passade bra ihop med hur läromaterialet skulle se ut då det betonade det jämnbördiga förhållandet mellan programkod och förklaringar. För att läromaterialet skulle vara lättförståeligt var det också viktigt att presentera materialet i den ordning som en mänsklig läsare, och inte datorn, tyckte var enklast. Avsnitt 2.3 beskriver hur litterat programmering fungerar i allmänhet och ger en bra bild av hur det såg ut även i detta projekt.

Under skrivandet av lärotexten lades övningar till. Dessa skapades genom att modifiera befintlig lärotext, istället för att förklara allting uppmanar den läsaren då och då att göra nästa steg i implementationen själv. När ett kapitel var avslutat lades dessutom extra

övningar till i slutet, dessa övningar var ofta vidareutvecklingar av det domänspecifika språk som redan implementerats.

## 3.2 Möten och återkoppling

För att få återkoppling på läromaterialet hölls ett kort och informellt möte med en testgrupp. Testgruppen bestod av tre andra studenter på Chalmers som gick tredje året på Datateknik och Informationsteknik. De hade alla klarat kursen Fysik för ingenjörer eller motsvarande och även klarat minst en kurs i Haskell. Däremot hade de inte läst DSLsof-Math eller motsvarande. Domänspecifika språk var med andra ord nytt för dem. Under mötet gavs läromaterialet en kort presentation och bakgrund. Sedan fick de på egen hand läsa materialet och deras spontana reaktioner och svar på frågor noterades.

Det hölls även två möten med Åke Fäldt. Ett möte hölls relativt tidigt i projektet, 2018-03-02, och ett andra relativt sent, 2018-04-11. Under mötena presenterades läromaterialet i sig och tanken med det, nämligen att presentera fysik ur ett annat perspektiv, ett programmeringsperspektiv. Diskussionen kretsade sedan kring svåra områden i Fysik för ingenjörer, vad läromaterialet skulle kunna bidra med för kunskaper till studenter samt dess eventuella roll i relation till fysikkursen.



# 4

## Resultat

I detta kapitel redovisas det resulterande läromaterialet, vilket består av fem kapitel. Det är publicerat på en hemsida och dess källkod är fritt tillgänglig. Även resultaten från mötena med testgruppen och Åke Fäldt redovisas.

### 4.1 Läromaterialet

Detta avsnitt innehåller en översikt av läromaterialet samt ett axplock av innehållet ur vardera kapitel. Axplocken exemplifier delar av läromaterialet och implementationerna av de domänspecifika språken. De fullständiga implementationerna är inte inkluderade (och förklarade) eftersom det är precis det läromaterialet innehåller. Rapporten skulle då bli en kopia av läromaterialet. Istället finns hela läromaterialet i bilaga A samt på hemsidan [16] där det blev publicerat. De avsnitt i rapporten som beskriver läromaterialets innehåll är inga exakta kopior ord-för-ord utan de har anpassats till rapporten.

#### 4.1.1 Översikt

Läromaterialet är en löpande text där Haskell-kod och lärotext sammanvävts. Det ser ut som i figur 4.1a. Lärotexten förklarar både fysik, Haskell och hur de relaterar till varandra. I texten följer läsaren med i implementationen av ett domänspecifikt språk och det visas hur det används. Tanken är att läsaren parallellt programmerar det som texten förklarar, för att på så sätt även få praktisk färdighet i det som presenterats. För detta syfte finns det även övningar tillagda direkt i den löpande texten, se figur 4.1b, som ofta innebär att läsaren själv ska implementera en liten del av det domänspecifika språket. Det finns också övningar i slutet av kapitlet som ofta innebär större vidareutvecklingar av de domänspecifika språken.

Språket i lärotexten är enligt projektgruppen lättsamt<sup>9</sup> och i detta syfte finns det även bilder tillagda. Figur 4.2 är ett exempel på en bild ur läromaterialet. Notera speciellt den medvetet oseriösa ritningstekniken som är tänkt att vara rolig och muntra upp läsaren.

---

<sup>9</sup>Diskuteras utförligare i avsnitt 5.2.

```
| Delta RealNum FunExpr
| D FunExpr
| I FunExpr
```

Even more finally, we add a deriving modifier to automatically allow for syntactic equality tests between FunExprs.

```
deriving Eq
```

**Exercise** Create values for acceleration, area and charge.

▼ **Solution**

```
acceleration = Dim 1 0 (-2) 0 0 0 0
area          = Dim 2 0 0 0 0 0 0
charge       = Dim 0 0 1 1 0 0 0
```

(a) Lärotexten är mot den ljusgrå bakgrunden medan Haskell-kod är mot den mörkgrå. (b) Exempel på en övning. Övningen ligger som en del av den löpande texten.

**Figur 4.1:** Ett smakprov av det resulterande läromaterialet.



**Figur 4.2:** Exempel på en bild. Bilden visar hur en hund springer och hoppar upp på en stillastående vagn.

Läromaterialet innehåller fem kapitel som vardera behandlar ett område inom fysik och matematik. Fokuset är på klassisk mekanik samt den matematik som tillhör området. De behandlade områdena är:

- Dimensioner
- Matematisk analys
- Vektorer
- Exempelproblem
- Partikelmekanik

*Dimensioner* behandlar fysikaliska dimensioner, storheter och enheter inom fysiken. Fysikaliska dimensioner införs på typnivå i Haskell för att visa likheten mellan Haskell's typsystem och hur dimensioner hanteras inom fysiken. Typnivåprogrammering<sup>10</sup> används för att göra likheterna så tydliga som möjligt.

I *matematisk analys* behandlas differentialekalkyl och integralkalkyl för en variabel. Först bestäms den semantiska domänen för analys i en variabel: reella funktioner av ett argument; och en datatyp för syntax för uttryck av funktioner inom denna domän konstrue-

<sup>10</sup>Vanligtvis manipuleras *värden* när programmering sker i Haskell och andra språk. Typnivåprogrammering är precis som vanlig programmering med skillnaden att den sker på typnivån, det vill säga, att typer modifieras. För en utförligare förklaring av typnivåprogrammering, se läromaterialet [16].

ras. Därefter analyseras syntax och semantik för differenser, derivator, och integraler; och funktioner implementeras för att utföra dessa operationer både approximativt numeriskt, och symboliskt med ett datatypen för syntaxen. Slutligen appliceras de implementerade funktionerna för att visualisera grafer av operationerna.

*Vektorer* behandlar vektorer och vektoroperationer. Vektorer modelleras med hjälp av en typklass som dikterar vilka funktioner som varje modell av en vektor måste implementera. Generella vektoroperationer såsom addition och skalärprodukt implementeras sedan med hjälp av dessa funktioner vilket skapade ett mycket generellt och lättanvänt gränssnitt. QuickCheck användes för att verifiera lagarna som gäller för olika vektoroperationer, vilket gav en generell säkerhet kring att implementationerna var korrekta.

I *exempelproblem* tillämpas de tre tidigare kapitlen på två vanliga mekanikproblem, nämligen *krafter på lådor* och *gungbräda*. I krafter på lådor används det domänspecifika språket för vektorer till att beräkna de krafter som verkar på en låda som glider ner för ett plan. I gungbräda visas hur momentjämviktsberäkningar kan göras med det domänspecifika språket för dimensioner.

*Partikelmekanik*. En repetition av gymnasieskolans fysik. Lägesenergi, rörelseenergi, gravitation och så vidare. Den modelleras med vektorer vars komponenter består av matematiska uttryck tagna från kapitlet om matematisk analys. Förhoppningen är att detta område inte ska presentera någon ny fysik för läsaren utan istället visa hur redan känd fysik direkt går att översätta till läromaterialets domänspecifika språk.

Läromaterialet blev publicerat på en hemsida [16] och all källkod finns tillgänglig på projektets GitHub [17]. På GitHub finns även ett antal delvis färdigställda områden, till exempel bevisföring. Texten i läromaterialet är skriven på engelska.

## 4.1.2 Dimensioner

I detta avsnitt visas delar av hur implementationen av dimensioner ser ut i läromaterialet. Den fullständiga implementationen innehåller tre delar:

1. Dimensioner på värdenivå
2. Dimensioner på typnivå
3. Datatyp för storheter

Dimensioner på värdenivå används för att enkelt kunna skriva ut dimensioner i GHCi. Dimensioner på typnivå används för att ge typsäkerhet till dimensioner, så att till exempel en längd och en massa inte kan adderas, liksom att ett värde av typ `Double` och `Integer` inte kan adderas i Haskell. Till sist kombineras de två varianterna av dimensioner till en datatyp för storheter som aritmetiska operationer kan utföras på.

Dimensioner kan ses som en produkt av de 7 basdimensionerna<sup>11</sup>, med en individuell exponent till varje basdimension. Datatypen, på värdenivå, som används ser därför ut

<sup>11</sup>Längd, massa, tid, elektrisk ström, temperatur, substansmängd och ljusstyrka.

som följande

```
data Dim = Dim Integer -- Length
           Integer -- Mass
           Integer -- Time
           Integer -- Current
           Integer -- Temperature
           Integer -- Substance
           Integer -- Luminosity

deriving (Eq)
```

Varje fält i datatypen representerar exponenten för motsvarande basdimension. Om exponenten är 0 betyder det att den basdimension inte är inkluderad i dimensionen. Några exempel ges för att förtydliga.

```
length      = Dim 1 0 0 0 0 0 0
mass        = Dim 0 1 0 0 0 0 0
time        = Dim 0 0 1 0 0 0 0

velocity     = Dim 1 0 (-1) 0 0 0 0
acceleration = Dim 1 0 (-1) 0 0 0 0
area         = Dim 2 0  0  0 0 0 0
```

Hastighet skrivs vanligtvis som  $\frac{m}{s}$  men ekvivalent är att skriva  $m^1 * s^{-1}$  vilket förklarar varför värdena ovan ser ut som de gör.

I resterande del av detta kapitel i läromaterialet visas hur multiplikation och division samt hur en *utskriftsfunktion*, som skriver ut ett värde snyggt, kan implementeras. Därefter följer ett antal delkapitel som innehåller testning, typnivådimensioner och storheter.

### 4.1.3 Matematisk analys

I kapitlet om matematisk analys skapas en syntax för funktionsuttryck och symbolisk derivering och integrering implementeras. Dessutom analyseras syntax och semantik hos uttryck som dyker upp inom matematisk analys, till exempel  $\Delta$ -operatoren.

Syntaxen för funktionsuttryck inleds med

```
data FunExpr = Exp | Log | Sin | Cos | Asin | Acos
```

vilket är några av de elementära funktionerna. Näst följer aritmetiska operationer. För att kunna definera dem på *funktioner* och inte algebraiska *uttryck* görs nedanstående tolkning

$$f \text{ OP}_{r \rightarrow r} g = x \mapsto (f(x) \text{ OP}_r g(x))$$

vilket utökar `FunExpr` med fler konstruktörer.

```
| FunExpr :+ FunExpr
| FunExpr :- FunExpr
| FunExpr :* FunExpr
```



```

| FunExpr :/ FunExpr
| FunExpr :^ FunExpr

```

Även konstruktörer för den oberoende variabeln, konstanter, funktionskomposition,  $\Delta$ , derivering och integrering behövs.

```

| Id
| Const RealNum
| FunExpr :. FunExpr
| Delta RealNum FunExpr
| D FunExpr
| I FunExpr

```

En utförligare förklaring av syntaxen `FunExpr` finns i läromaterialet.

I läromaterialet analyseras derivering utförligt med gränsvärden, kvoter och approximativa metoder. Dessutom motiveras och implementeras symbolisk derivering. Den görs med en funktion `derive` som har ett fall för vardera av konstruktörerna i `FunExpr`. Några av fallen är

```

derive (f :* g) = derive f :* g :+ f :* derive g
derive (f :+ g) = derive f :+ derive g
derive Exp = Exp

```

#### 4.1.4 Vektorer

Kapitlet om vektorer inleds med en genomgång av vektorer och operationer på dem parallellt med att en första implementation till dem skrivs. Implementation ser ut som följer

```

data Vector2 n = V2 n n
type Scalar = Double
type VectorTwo = Vector2 Scalar

magnitude :: VectorTwo -> Scalar
magnitude (V2 x y) = sqrt (x^2 + y^2)

add :: VectorTwo -> VectorTwo -> VectorTwo
add (V2 x1 y1) (V2 x2 y2) = V2 (x1 + x2) (y1 + y2)

dotProd :: VectorTwo -> VectorTwo -> Scalar
dotProd (V2 ax ay) (V2 bx by) = ax * bx + ay * by

```

Eftersom vektorer i tre dimensioner ser snarlik ut leder det till väldigt mycket duplicerad kod. En andra implementation görs därför som är mer generell. Grunden är den nedanstående typklassen

```

class Vector vector where
  vmap      :: (n -> n) -> vector n -> vector n
  vzipWith :: (n -> n -> n) -> vector n -> vector n
                                     -> vector n

```

```
    vfold      :: (n -> n -> n) -> vector n -> n
```

som innehåller de tre funktioner som behövs för att genomföra vektoroperationerna `magnitude`, `add` och `dotProd`. De kan nämligen skrivas om enbart med hjälp av de funktioner som finns i typklassen `Vector` på följande sätt

```
magnitude :: (Floating n, Vector vec) => vec n -> n
magnitude = sqrt . vfold (+) . vmap (**2)
```

```
add :: (Num n, Vector vec) => vec n -> vec n -> vec n
add = vzipWith (+)
```

```
dotProd :: (Num n, Vector vec) => vec n -> vec n -> n
dotProd v1 v2 = vfold (+) $ vzipWith (*) v1 v2
```

Allt som behöver göras för vardera datatyp för vektorer av olika längd är därmed att göra den en `Vector`-instans. Så här ser det ut för `Vector2`

```
instance Vector Vector2 where
    vmap      f (V2 x y)           = V2 (f x)      (f y)
    vzipWith f (V2 x y) (V2 x' y') = V2 (f x x') (f y y')
    vfold    f (V2 x y)           = f x y
```

Resterande del av vektor-kapitlet behandlar kryssprodukt, vektorlagar och tester av dem.

### 4.1.5 Exempelproblem

Dessa problem är implementationer eller lösningar på fysikuppgifter som har förekommit på tentamen i Fysik för ingenjörer. De använder de andra modulerna i läromaterialet för att göra beräkningarna (exempelvis dimensionsanalys eller vektoroperationer). Av exempelproblemen finns en gungbräda och en låda på ett lutande plan. Då källkoden för dessa problem endast är menade att lösa den specifika uppgiften, och inte alla varianter på gungbrädor eller lådor på lutande plan, så kan de inte kallas för domänspecifika språk, utan är istället implementationer som använder tidigare domänspecifika språk.

Exemplet med gungbrädan har två massor på gungbräda som befinner sig i jämvikt. Uppgiften är att finna en okänd sträcka mellan en av vikterna och balanspunkten. Exemplet använder dimensionsanalys för att representera hävarmseffekten av de olika massorna. Genom att ställa upp förhållanden och lösa ut den okända sträckan, så går det att beräkna den okända sträckan. Sist görs en säkerhetskontroll där den okända sträckan används för att räkna ut hävarmsmomenten, vilket är den sista kontrolleringen att man inte har gjort några logiska fel.

Exemplet med en låda på ett lutande plan undersöker krafterna på en låda på ett lutande plan, såsom gravitationen, normalkraften från det lutande planet, och resultantkraften, vilka representeras genom vektorer från vektormodulen. Detta exempel tar både hänsyn till statisk friktion (då lådan befinner sig stillastående) och kinetisk friktion (då lådan befinner sig i rörelse).

### 4.1.6 Partikelmekanik

Implementationen av partikelmekanik är en kombination av de domänspecifika språken för vektorer och matematisk analys. Anledningen till detta är att partiklars position, hastighet och acceleration modelleras med vektorer, dessutom är de krafter som påverkar partiklar även de modellerade som vektorer. Sedan används matematisk analys för att göra beräkningar på dem. Därför är det naturligt att modellera partikelmekanik med hjälp av vektorer vars komponenter är uttryck som implementeras av matematisk analys.

Datatypen för en partikel är

```

type Mass      = FunExpr
type VectorE  = Vector3 FunExpr

data Particle  = P { pos    :: VectorE -- Position as a
                                -- function of time.
                                -- Unit m
                        , mass :: Mass   -- Mass. Unit kg
                        }

```

Ett exempelvärde är

```

particle :: Particle
particle = P (V3 (3 * Id * Id) (2 * Id) 1) 3

```

Eftersom komponenterna i vektorerna är funktionsuttryck är det enkelt att beräkna en partikels hastighet och acceleration.

```

velocity :: Particle -> VectorE
velocity = vmap D . pos

acceleration :: Particle -> VectorE
acceleration = vmap D . velocity

```

Det går även att beräkna nettokraften på en partikel med hjälp av Newtons andra lag.

```

type Force = VectorE

force :: Particle -> Force
force p = vmap (* m) a
  where
    m = mass p
    a = acceleration p

```

I läromaterialet behandlas även rörelseenergi, lägesenergi, arbete och gravitation.

## 4.2 Möten och återkoppling

Återkopplingen från testgruppen var till övervägande del positiv. Testgruppen tyckte läromaterialet var ett intressant och roligt sätt att presentera fysik på. De tyckte att

bilderna tjänade sitt syfte i att muntra upp läsaren. Mötet var dock för kort för att det skulle framgå huruvida läsaren lärde sig mest fysik eller mest Haskell. Det framgick heller inte om läromaterialet uppmuntrade testgruppen att vilja lära sig mer fysik.

Åke Fäldt hade en överlag positiv syn på läromaterialet<sup>12</sup>. Fäldt tyckte att det fanns flera saker läromaterialet kunde bidra med. En var att läromaterialet ger ett annat perspektiv på fysiken, en annan var den rigorositet som domänspecifika språk innefattar. Eftersom de domänspecifika språken måste vara väldefinierade betyder det att alla fysikaliska koncept måste göras entydiga och väldefinierade. Följden blir att operationerna enbart kan göras på det definierade sättet. Dessutom måste dimensionerna stämma. Fäldt menade att detta var bra egenskaper med läromaterialet och att det rigorösa tankesätt samt den metodik som förmedlas hade varit till nytta för problemlösning i fysikkursen. Fäldt nämnde även områden i Fysik för ingenjörer som var svåra för studenter, vilket användes vid sökandet efter områden, se avsnitt 3.1.1.

Roger Johansson, programansvarig på datateknik, och datas nämnd för studier (DNS) var båda positiva till projektets initiativ och målsättning. Dessa möten var viktiga framförallt för att göra de som styr utbildningen på datasektionerna medvetna om projektets existens.

---

<sup>12</sup>Det bör påpekas att det som är återgivet här självklart är en egen tolkning, och kan ha missuppfattats, av projektgruppen. Fäldt ska med andra ord inte behöva stå till svars för vad som står här.

# 5

## Diskussion

I detta kapitel diskuteras projektets genomförande, resultat, vidareutvecklingsmöjligheter och etiska aspekter.

Till att börja med vill vi säga att hur domänspecifika språk kan kombineras med fysik inte var något vi visste när projektet startade. En stor del av arbetet i början av projektet ägnades därför åt att försöka komma på olika sätt att använda dem ihop med olika fysikaliska områden. Det gjordes många experiment innan vi hittade ett sätt att skapa domänspecifika språk för fysik som var annat än triviala implementationer av formler, till exempel att formeln för rörelseenergi,  $E_k = \frac{mv^2}{2}$ , kan skrivas som `ek m v = m * v * v / 2` i Haskell. Detta experimenterande ledde till att vi såg en slags strategi för hur de kan kombineras, vilket blev den metodik som beskrivs i avsnitt 3.1. Det vi vill poängtera är med andra ord att det har varit oklart hur projektet skulle kunna föras framåt eftersom det inte funnits någon tydlig väg att följa.

### 5.1 Genomförandediskussion

Under projektets genomförande har det gjorts flera val av teorier och metoder att använda. Självklart behöver inte dessa val vi gjorde vara de bästa. Därför kommer vi här att kritisera dem och föreslå andra möjligheter. Närmare bestämt kommer mötet med testgruppen, urvalet och Literate Haskell att diskuteras.

Det sätt återkopplingen gjordes under projektet kan kritiseras på flera sätt. För det första bestod testgruppen av enbart tre personer. För det andra hölls mötet under en ytterst kort tid, ungefär en timme. Mötet hade behövt vara längre för att låta testgruppen i lugn och ro arbeta igenom ett par kapitel, inklusive att följa med i programmeringen som gjordes i läromaterialet. För det tredje var vi inte tydliga med målet med återkopplingen, nämligen om de tyckte det var meningsfullt att lära ut fysik med hjälp av domänspecifika språk. Det gjorde att de heller inte kunde tänka på dessa frågor. Med tanke på dessa tre brister i återkopplingens genomförande är alla slutsatser dragna med den som stöd ytterst osäkra.

Det går även att kritisera hur urvalet av områden gick till under projektet. Dels kan det ha lett till att enbart två domänspecifika språk för fysik implementerades (dimensioner

och partikelmekanik). Dels skedde urvalet ur implementatörens perspektiv (det vill säga vårt) och inte ur användarens perspektiv (studenten som ska nyttja läromaterialet). Med det menar vi att områden valdes utifrån hur det implementationsmässigt hängde ihop, till exempel att matematisk analys är grunden till flera tillämpningar. Istället hade områden kunnat väljas utifrån de fysikaliska problem studenter ska lösa i Fysik för ingenjörer, till exempel block och talja eller momentjämvikt, och utifrån det utforma domänspecifika språk.

Till sist kan vi kritisera den allmänna metoden som valdes för utformningen av läromaterialet, nämligen att skriva varje kapitel som en löpande text om hur ett domänspecifikt språk implementeras eller tillämpas. Nackdelen är att det blir en passiv inläring, även om det till en viss utsträckning finns övningar och läsaren uppmuntras implementera koden parallellt. Literate Haskell har definitivt bidragit till dessa passiva tendenser. Literate Haskell är en programfil som är väldigt väl beskriven hur programmet fungerar, vilket leder till att man ser den som en text att läsa istället för ett program att köra och experimentera med. Under projektets genomförande hade det därför varit av intresse att undersöka alternativa sätt att utforma lärotexten som uppmuntrat ett mer aktivt lärande. Det hade till exempel kunnat vara att presentera idén bakom fysikaliska dimensioner, och sedan låta studenten själv skapa implementationen. En fingervisning i form av ett enkelt exempel på en möjlig lösning kan vara ett sätt att göra detta.

## 5.2 Resultatdiskussion

Detta avsnitt inleds med en övergripande diskussion om det resulterande läromaterialet och för vem läromaterialet kan vara relevant, för att sedan övergå till en mer generell diskussion kring kombinationen av domänspecifika språk och fysik.

I projektets mål och avgränsningar stod det att vi skulle börja med klassisk mekanik, för att i mån av tid även behandla termodynamik och vågrörelselära. I avsnitt 4.1 nämns att de tre grundläggande områdena dimensioner, matematisk analys och vektorer är färdiga, samt de komposita områdena partikelmekanik, gungbräda och krafter på lådor. Med andra ord har mekanik påbörjats, men inte termodynamik eller vågrörelselära. Det som återstår enligt oss när det kommer till mekanik är att tillämpa de grundläggande områdena på fler fysikaliska problem utöver gungbräda och krafter på lådor. Vi tror att de tre grundläggande områdena som är färdiga räcker. Förutom fler tillämpningar kan mer fördjupande områden implementeras, till exempel bevisföring, något som redan påbörjats, se avsnitt 4.1.

En annan del av målet var att läromaterialet skulle vara lättillgängligt genom sitt språkbruk, publicering på en hemsida och fri tillgång till källkoden. Vi kan konstatera att de senare två har genomförts. Vi passar även på att säga att vi tycker att en lättanvänd hemsida är trevligare att använda än PDF-filer eftersom de inte har sidbrytningar, fasta sidmarginaler som använder skärmutrymme, med mera. Detta är visserligen mindre detaljer, men tillsammans påverkar de upplevelsen i stort. Vi tycker även att språket i läromaterialet är någorlunda lättsamt då vi skriver talspråkligt och vardagligt, och för-

klaras svårigheterna grundligt. Språket hade dock kunnat vara vänligare. Till exempel beskriver vi olika koncept som “våldigt enkla” fastän läsaren kanske inte alls tycker det.

Vi knyter här även an till lärandeteorierna i avsnitt 2.4, som nämnde interaktion och snabba belöningar. Vårt läromaterial har visserligen ingen interaktiv sida, men typsyste- met i Haskell kan ändå fungera som en fingervisare när programmeraren gör rätt eller fel. Det går exempelvis inte att räkna med dimensioner på ett felaktigt sätt, och funktions- komposition fungerar endast om båda funktionernas typdefinitioner (typer på argument och returvärde) stämmer överens. När det kommer till snabba belöningar kan den glädje programmeraren ser när koden kompilerar ses som en sådan. Läromaterialet innefattar även strategiskt placerade roliga bilder för att ge impulsiva glädjereaktioner.

Läromaterialet kan vara relevanta för flera grupper. Visserligen är målgruppen datastu- denter, och vi har personligen dragit nytta av det, men vi tror att det kan vara relevant för fler än så, till exempel kan läromaterialet även vara intressant för fysiklärare. Fäldt nämnde att han tyckte att det rigorösa tankesätt läromaterialet skolar in läsaren i kan vara användbart även i traditionell fysikundervisning. Det kan därför vara intressant att undersöka hur ett sådant här läromaterial kan integreras i undervisningen.

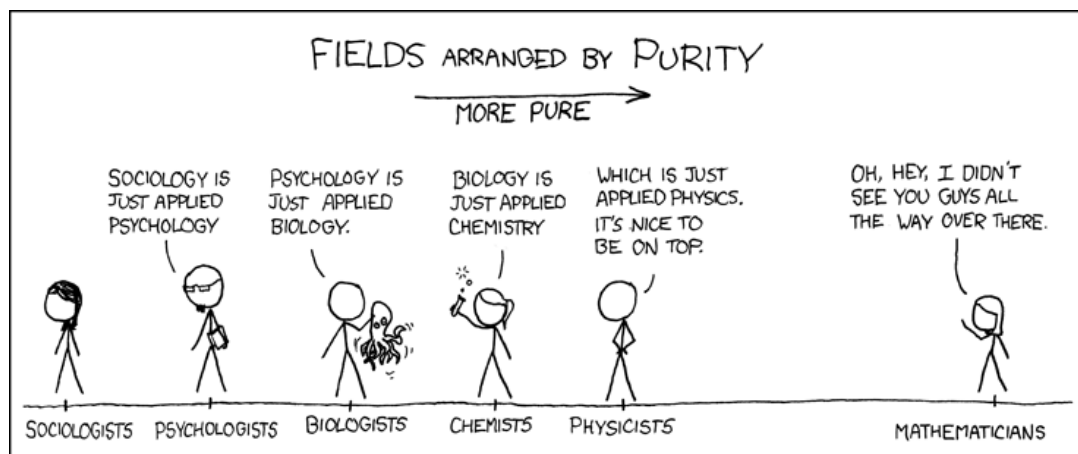
En del av projektets mål var att diskutera kombinationen av domänspecifika språk och fysik. I de tre följande avsnitten diskuterar vi därför läromaterialets fokus på matematik och Haskell istället för fysik, vilka fysikaliska områden som var lämpliga att göra do- mänspecifika språk till samt om det finns en pedagogisk nytta i att kombinera de två. Diskussionen är till största del baserad på våra erfarenheter efter att ha implementerat ett flertal domänspecifika språk relaterade till fysik, men de är även baserade på uppfatt- ningar från testgruppen och Fäldt.

### 5.2.1 Om läromaterialets fokus på matematik och Haskell snarare än fysik

Målet med läromaterialet var att lära ut fysik på ett roligt och lättförståeligt sätt. Dock syns det i resultatet, avsnitt 4.1, att mest fokus lagts på att förklara och lära ut matematik och Haskell. Det finns flera anledningar till varför så är fallet.

Problemet med att prata om en egen implementation av fysik är att fysik inte är ett helt eget område. Snarare kan fysik betraktas som en kombination av tillämpad matematik, experimentering, och problemlösning. Det blir därför naturligt att när den faktiska im- plementationen av fysikaliska beskrivningar och lagar sker, så är matematiken i centrum. Att fysik är tillämpad matematik illustreras i figur 5.1, som även visar mer generellt hur en kedja av områden är tillämpningar av varandra.

Anledningen till att stor fokus ibland läggs på Haskell är att de koncept som används är viktiga för läsaren att förstå, men vi vill förutsätta så lite förkunskaper som möjligt så att även oerfarna Haskell-programmerare ska kunna följa med i boken. Ett syfte med läro- materialet är också att väcka intresse hos läsaren, och vi tror att detta kan uppnås genom att visa paralleller mellan funktionell programmering, matematik, och implementationen



**Figur 5.1:** Purity (C) Randall Munroe ( xkcd.com) CC BY-NC. Bilden beskriver hur fysik är en tillämpning av matematik. Det är i själva verket så att en kedja av områden kan betraktas som tillämpningar av varandra.

av fysik.

Hade andra val gjorts i projektets tidigare skede, hade det kanske sett annorlunda ut. Tidigt valde vi att söka efter områden som vi ansåg vara fristående och väl avgränsade, se avsnitt 3.1.1, och implementera dessa var för sig. Detta sätt att påbörja projektet har påverkat allting som kom därefter. Om vi istället hade utvecklat läromaterialet som en kombination av olika områden från början hade det kanske inte varit lika främmande att även baka in problemlösning i läromaterialet och på så sätt fått mer fysikorienterade domänspecifika språk. Något annat som kan ha påverkat fokuset är valet av Haskell som implementeringsspråk. Vi hävdar i teorikapitlet 2.2 att Haskells typer och fokus på mönstermatchning gör det idealt för implementering av domänspecifika språk, men medför inte nödvändigtvis att det är idealt för implementering av fysik. Det är möjligt att ett annat språk eller till och med en annan paradigm hade lett till att fokus hade hamnat på fysiken framför matematiken.

## 5.2.2 Lämpliga områden för domänspecifika språk

När domänspecifika språk implementerades visade det sig att vissa områden lämpade sig bättre än andra. Exempel på ett lämpligt och ett mindre lämpligt område är vektorer respektive lutande plan<sup>13</sup>. Vad som skiljer dem åt är enligt oss att lämpliga områden består av tydliga *data och operationer* medan mindre lämpliga områden består av *egenskaper och samband*.

Inom området vektorer är vektorer data och till exempel vektoraddition en operation. I

<sup>13</sup>Läromaterialet behandlar fortfarande lutande plan, men inte som ett eget domänspecifikt språk. Istället *tillämpas* det domänspecifika språket för vektorer.



matematik defineras (en tvådimensionell) vektor som

$$\vec{v} = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$$

och vektoraddition som

$$\vec{u} + \vec{w} = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} + \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} u_1 + w_1 \\ u_2 + w_2 \end{bmatrix}$$

I Haskell kan en datatyp för vektorer defineras som

```
data V = V Double Double
```

och vektoraddition som

```
va :: V -> V -> V  
(V u1 u2) 'va' (V w1 w2) = V (u1+w1) (u2+w2)
```

Det finns enligt oss en tydlig likhet mellan matematik och Haskell i detta fall, och även i för andra lämpliga områden, vilket gör att vi tycker det blir enkelt att modellera och förstå områden som vektorer.

Egenskaper och samband är enligt oss svårare att modellera meningsfullt. Visserligen kan ekvationer modelleras i Haskell, men då blir det bara en ekvationslösare. Nyckeln, i till exempel lutande plan, är att känna till sambanden och veta när de ska användas. Vi behandlar därför lutande plan genom att tillämpa det domänspecifika språket för vektorer. Att vissa områden var mindre lämpliga var ett oväntat resultat. I början av projektets start trodde vi att domänspecifika språk skulle kunna implementeras till alla typer av områden.

### 5.2.3 Domänspecifika språk, fysik och pedagogiska aspekter

En del av projektets mål är att diskutera huruvida det finns en pedagogisk nytta i att kombinera fysik och domänspecifika språk. Denna fråga diskuteras nedan.

I samband med fysik finns det några fördelar med att integrera domänspecifika språk. Domänspecifika språk kan betraktas som “tools for thinking”<sup>14</sup> och ger ett nytt perspektiv på fysik och ger den *struktur*. Dimensioner är ett exempel på detta, se avsnitt 4.1.2. Där konstateras att en godtycklig dimension kan skrivas som de sju basdimensionerna med tillhörande exponenter, vilket kanske inte är så man brukar se på dimensioner, men som ger dem en väldefinierad struktur. Domänspecifika språk bidrar även med *rigorösitet* till fysik. Enbart de definierade operationerna går att använda, vilket leder till att genvägar i fysikaliska beräkningar inte går att göra på det sätt som är möjligt vid räkning med papper och penna. Detta tyckte även Åke Fäldt var en bra aspekt, se avsnitt 4.2. Med hjälp av domänspecifika språk är det dessutom möjlighet att väcka *intresse* för fysik. En student som inte är intresserad av fysik kanske skulle bli det om fysik presenteras i

<sup>14</sup>Uttryckt i Patrik Janssons egna ord, föreläsare i kursen DSLsofMath.

samband med Haskell och domänspecifika språk, där paralleller mellan dem visas. Denna tanke stöds även av testgruppen<sup>15</sup>, se avsnitt 4.2

År 2016 genomfördes ett kandidatarbete på Chalmers liknande detta [5]. Det kandidatarbetet resulterade också i ett läromaterial, skillnaden är att det arbetet behandlade signallära. Grundidén är dock densamma: att använda domänspecifika språk för att ge struktur till ett annat område. Dessutom finns kursen *Classical Mechanics: A Computational Approach* [3] som också använde domänspecifika språk till att ge struktur till fysik. Detta tycker vi visar på att det finns ett akademiskt intresse för att använda domänspecifika språk i syfte att lära ut, och att idén som vi presenterar i denna rapport även går att applicera på andra områden. Nyttan att strukturera upp områden i väl avgränsade och tydligt definierade delar kanske anses som uppenbar, men frågan om vilket verktyg som ska användas för detta är inte lika uppenbar. Vi hävdar att domänspecifika språk är ett sådant verktyg.

Det finns dock problem med att använda domänspecifika språk till att lära ut fysik. Ett problem är att den kreativa problemlösning som ingår i fysik är svår att fånga upp i domänspecifika språk<sup>16</sup>. Ett annat problem är att det finns risk att fokus glider från fysik till domänspecifika språk. Ett läromaterialet om renodlad fysik med ett lättsamt språk och noggrann förklaring av koncepten hade säkert varit mycket uppskattat, det sätt som Khan Academy [18] beskriver fysik är ett sådant exempel. En större målgrupp hade också kunnat nå då.

Avslutningsvis tror vi att domänspecifika språk med fördel kan integreras i traditionell fysikundervisning och på så sätt fånga upp de delar domänspecifika språk gör bra i fysik: strukturera, uppmuntra rigorös problemlösning och väcka intresse. Men samtidigt förlita sig på traditionella metoder till stor del i bland annat problemlösning och inte låta fokuset glida för långt från fysik till domänspecifika språk.

### 5.3 Vidareutvecklingsmöjligheter och behov av ytterligare kunskap

Läromaterialet innehåller domänspecifika språk för de *matematiska* områdena analys och vektorer. Dessa områden används sedan för att koda upp och lösa uppgifter av mer *fysikaliska* slag, till exempel krafter på lådor. Med andra ord hanteras fysikaliska områden genom att *tillämpa* matematiska domänspecifika språk och inte genom att *konstruera* fysikaliska domänspecifika språk. En vidareutveckling hade därmed varit att göra precis det, att inte tillämpa matematiska domänspecifika språk utan att göra fysikaliska domänspecifika språk. Det kan till exempel vara saker som ett språk för ett lutande plans komponenter, ett språk för vilka krafter som verkar på fysikaliska kroppar i mekanikproblem eller till och med ett språk för något så abstrakt som fysikalisk problemlösning i allmänhet. Vi vet inte hur ett domänspecifikt språk av detta slag kan se ut, vilket är anledningen till att vi gick den andra vägen, som vi diskuterade i avsnitt 5.2.1. Att göra

---

<sup>15</sup>Eftersom mötet med testgruppen var väldigt kort är det dock svårt att dra några säkra slutsatser.

<sup>16</sup>Enligt oss, och vi vet inte hur det skulle kunna göras.

fler fysikorienterade domänspecifika språk hade därför varit en möjlig vidareutveckling.

En annan möjlig vidareutveckling är att göra en rigorös studie kring de pedagogiska aspekterna hos kombinationen av fysik och domänspecifika språk. Detta projekt innehöll enbart en mindre sådan studie. Det som kan vara intressant att undersöka är om studenter tycker att fysik blir intressantare genom en kombination av detta slag och kanske därför studerar mer i fysikkursen. Det hade också varit intressant att undersöka om det rigorösa tankesätt domänspecifika språk förmedlar (se avsnitt 5.2.3) spiller över och gör nytta inom traditionell fysikundervisning. Det är för dessa två frågor det främsta behovet av ytterligare kunskaper ligger. Ett av målen med detta projekt var trots allt att förbättra fysikkunskaper (genom ökat intresse eller mer rigorösitet) och då är det viktigt att undersöka om det faktiskt blir så i praktiken.

Även det befintliga läromaterialet kan byggas vidare på. I sin nuvarande form behandlas varken termodynamik eller vågrörelselära alls. Dessutom finns det aspekter inom den klassiska mekaniken som fattas.

Slutligen finns det en mycket intressant vidareutveckling som inte alls har behandlats i detta projekt, nämligen att använda matematiska och fysikaliska domänspecifika språk som ett syntaktiskt lager mellan användare och en underliggande komplex kodbas. I många fall kan dessa kodbaser vara implementerade på sätt som inte förefaller vara intuitivt, och saknar typsäkerhet. I dessa fall kan det vara användbart med ett domänspecifikt språk med hög typsäkerhet som tvingar användaren att endast skriva korrekta uttryck och som döljer den bakomliggande komplexiteten. Tänk till exempel på kod i Matlab. Där är det lätt hänt att missa någon detalj i sin implementation så att beräkningarna blir fel utan att Matlab klagar. Uttrycket är korrekt men semantiken är fel. Med ett syntaktiskt lager som kräver att de fysikaliska dimensionerna stämmer överens hade vissa misstag kunnat upptäckas vid kompilering istället för att kanske inte upptäckas alls. Denna idé framfördes till oss av Jeff Chen<sup>17</sup> på Chalmers, men är även någonting som har genomförts på andra områden, till exempel inom molekylär dynamik [19].

## 5.4 Etiska aspekter

En bakomliggande tanke vi haft genom hela projektet är att läromaterialet ska vara fritt tillgängligt för alla, därför har vi valt att publicera det på en hemsida. Denna hemsida använder grundläggande HTML och CSS samt javascript. Javascript är dock inget strikt krav för funktionaliteten. Eftersom hemsidan är lättviktig bör den fungera väl även på gamla datorer och telefoner, till skillnad från tunga PDF-filer och många moderna hemsidor. Läromaterialet blir på detta sätt tillgängligt även för studenter i länder med sämre internetuppkoppling.

Tanken om tillgänglighet ligger även bakom valet att låta källkoden vara fritt tillgänglig. Visserligen *är* läromaterialet i princip hela källkoden, det vill säga har läsaren läromaterialet har den källkoden. Men att ha tillgång till källkoden direkt har dock fördelar som

---

<sup>17</sup>Jeffs sida på Chalmers: <http://www.cse.chalmers.se/~yuting/>.

att läsaren kan följa med i versionshistoriken, se kommentarer och alternativa implementationer som inte syns i den slutgiltiga produkten samt att det blir enklare att modifiera källkoden och lära sig om hemsidans uppbyggnad. Det handlar om transparens, att visa att skaparen är positiv till granskning och till att låta andra bygga vidare på ens skapelser. Genom att sluta oss till skaran som skapar öppen källkod hoppas vi att fler inom samhället i stort ska gå över till denna modell.

Valet att skriva på engelska har också att göra med tillgängligheten. Fler kan engelska än svenska. På detta sätt kan läromaterialet komma fler till gagn.

En möjlig negativ etisk aspekt är förstås att enbart studenter som kan Haskell kan dra nytta av detta läromaterial. Om det skulle integreras i traditionell fysikundervisning som ett frivilligt extramaterial skulle vissa studenter få sämre möjligheter att lära sig. Även om en introduktion till Haskell inkluderats i fysikkursen är det tveksamt hurvuda de studenter som inte kunde Haskell sedan innan skulle kunna tillgodogöra sig innehållet.

# 6

## Slutsatser

Projektets mål var att konstruera ett läromaterial som modellerar fysik med hjälp av domänspecifika språk samt att diskutera hur väl det fungerar och om det finns en pedagogisk nytta i det. Bakgrunden låg i att projektgruppen ville väcka intresse för fysik hos datastudenter genom att presentera det ur ett programmeringsperspektiv. Det skulle förhoppningsvis kunna förbättra den mindre bra tentastatistiken i kursen Fysik för ingenjörer.

Resultatet blev ett läromaterial bestående av domänspecifika språk i Haskell sammanvävt med en lärotext som förklarar dem. Det innehåller kapitel för dimensioner, matematisk analys, partikelmekanik, vektorer och tillämpningar av dem på fysikaliska problem. Det ingår även programmeringsövningar. Läromaterialet publicerades på en hemsida som är fritt tillgänglig för alla att besöka.

För att undersöka den pedagogiska nyttan hölls två möten med Åke Fäldt, föreläsare och examinator för Fysik för ingenjörer. Det framgick från mötena med Fäldt att det fanns en klar pedagogisk nytta med det tillvägagångssätt som vi använt för att beskriva fysik. Den rigorösa struktur som Haskell tvingar en att använda ger inte utrymme för en student att hoppa över delar av förståelsen för ett fysikproblem. Istället tvingas studenten att implementera hela lösningen från grunden och detta kan ge en djupare förståelse för problemet och fysiken i stort.

Det faktiska resultatet från detta tillvägagångssätt, läromaterialet, testades av en testgrupp. Från denna grupp fick vi mycket positiv kritik och de tyckte att läromaterialet var pedagogiskt, roligt och intressant. Men för att verkligen kunna dra några slutsatser från detta krävs en mycket mer noggrann undersökning huruvida studenter blir bättre på fysik med hjälp av ett läromaterial av detta slag.

Till sist vill vi säga vi anser att det material vi har producerat har följt våra mål väl eftersom det är roligt, lättläst och intressant. Och även om den pedagogiska nyttan av materialet inte är ordentligt testat tycker vi alla i projektgruppen att vi har dragit stor nytta av att utveckla det.



# Litteraturförteckning

- [1] G. Johansson, “Statistik över kursresultat”, 2018. [Online]. Tillgänglig: <http://document.chalmers.se/doc/00000000-0000-0000-0000-00001C968DC6>, hämtad: 2018-01-30.
- [2] C. Ionescu och P. Jansson, “Domain-Specific Languages of Mathematics: Presenting Mathematical Analysis Using Functional Programming”, i *Proceedings of the 4th and 5th International Workshop on Trends in Functional Programming in Education*, Sophia-Antipolis, France and University of Maryland College Park, USA, 2015, ss. 1-15. [Online]. Tillgänglig: <https://doi.org/10.4204/EPTCS.230.1>, hämtad: 2018-01-30.
- [3] G. J. Sussman och J. Wisdom, “Classical Mechanics: A Computational Approach”, 2008. [Online]. Tillgänglig: <https://ocw.mit.edu/courses/earth-atmospheric-and-planetary-sciences/12-620j-classical-mechanics-a-computational-approach-fall-2008/>, hämtad: 2018-01-30.
- [4] G. J. Sussman, J. Wisdom och M. E. Mayer, *Structure and Interpretation of Classical Mechanics*. Cambridge, MA, USA: MIT, 2001. [Online]. Tillgänglig: <https://mitpress.mit.edu/sites/default/files/titles/content/sicm/book.html>, hämtad: 2018-03-07.
- [5] F. Lindahl, C. Rosvall, P. Ngo, J. Jonsson och J. Olsson, “Programmering som undervisningsverktyg för Transformer, signaler och system: Utveckling av läromaterialet TSS med DSL”, Chalmers tekniska högskola, Göteborg, Sverige, 2016.
- [6] S. N. Walck, “Learn Physics by Programming in Haskell,” 2017. [Online]. Tillgänglig: <https://arxiv.org/abs/1412.4880v1>, hämtad: 2018-05-14.
- [7] D. Ghosh, “DSL for the Uninitiated”, *Queue*, vol. 9, nr. 6, s. 10-21, jun. 2011. [Online]. Tillgänglig: <https://doi.org/10.1145/1989748.1989750>, hämtad: 2018-03-07.
- [8] D. Knuth, “Literate Programming”, *The Computer Journal*, vol. 27, nr. 2, ss. 97-111, jan. 1984. [Online]. Tillgänglig: <https://doi.org/10.1093/comjnl/27.2.97>, hämtad: 2018-03-14.
- [9] Haskell Wiki, “Literate Programming”, 2017. [Online]. Tillgänglig: [https://wiki.haskell.org/Literate\\_programming](https://wiki.haskell.org/Literate_programming), hämtad: 2018-03-14.

- [10] Pandoc, “Pandoc: a universal document converter”, 2018. [Online]. Tillgänglig: <https://pandoc.org/>, hämtad: 2018-03-14.
- [11] J. M. Keller, *Motivational Design for Learning and Performance: The ARCS Model Approach*. New York, New York, USA: Springer, 2010.
- [12] U. P. Lundgren, R. Säljö, C. Liberg (Red.), “Lärande Skola Bildning”, 2 uppl., Stockholm, Natur & Kultur, 2012.
- [13] A. Rozental, L. Wennersten, “Dansa på deadline: Uppskjutandets psykologi”, 1 uppl., ss. 41, Stockholm, Natur & Kultur, 2014.
- [14] H. D. Young och R. A. Freedman, *University Physics with Modern Physics*, Harlow, England: Pearson Education Limited, 2015.
- [15] Hackage, “QuickCheck: Automatic testing of Haskell programs”, 2018. [Online]. Tillgänglig: <https://hackage.haskell.org/package/QuickCheck>, hämtad: 2018-04-26.
- [16] B. Werner, E. Sjöström, J. Johansson och O. Lundström, “Learn You a Physics for Great Good!”, 2018. [Online]. Tillgänglig: <https://dslsofmath.github.io/BScProj2018/>, hämtad: 2018-03-14.
- [17] B. Werner, E. Sjöström, J. Johansson och O. Lundström, “BScProj2018”, 2018. [Online]. Tillgänglig: <https://github.com/DSLsofMath/BScProj2018>, hämtad: 2018-03-14.
- [18] Khan Academy, “Khan Academy | Free Online Courses, Lessons & Practise”, 2018. [Online]. Tillgänglig: <https://www.khanacademy.org/>, hämtad: 2018-04-11.
- [19] E. H. Müller, J. Grant och W. R. Saunders, “A Domain Specific Language for Performance Portable Molecular Dynamics Algorithms,” 2017. [Online]. Tillgänglig: <https://arxiv.org/abs/1704.03329>, hämtad: 2018-04-20.



# A

## Läromaterialet

Denna bilaga innehåller en ögonblicksbild av hur läromaterialet såg ut 2018-05-30.

# Learn You a **Physics** for Great Good!

>>> WORK IN PROGRESS <<<

## 1. Introduction

---

- [About this book](#)
- [So what's a DSL?](#)
- [Getting started](#)

## 2. Calculus

---

- [Introduction](#)
- [Function expressions](#)
- [Differential calculus](#)
- [Integral calculus](#)
- [Plotting graphs](#)
- [Syntax trees](#)

## 3. Linear algebra

---

- [Vectors](#)

## 4. Dimensions

---

- Introduction
- Value-level dimensions
- Testing of value-level dimensions
- Type-level dimensions
- Quantities
- Testing of Quantities
- Usage

## 5. Newtonian Mechanics

---

- Single particle mechanics

## 6. Examples

---

- Teeter
- Box on an incline

# Learn You a **Physics** for Great Good!

>>> WORK IN PROGRESS <<<

## Introduction / About this book

[src: [Introduction/About.lhs](#)] Previous: [Table of contents](#) [Table of contents](#) Next: [So what's a DSL?](#)

## About this book

You've arrived at **Learn You a Physics for Great Good**, the #1 stop for all your beginner-to-intermediate needs of learning **Physics** through the use of **Domain Specific Languages** (DSLs).

This book was written as a [BSc thesis project](#) at Chalmers University of Technology as an offshoot of a bachelor's level elective course [Domain Specific Languages of Mathematics](#). The goal of the the project and the reason for the existence of this book, is to help (primarily CS) students learn physics better. We think that the use of domain specific languages to teach the subject will help set these students in the right mindset to learn physics efficiently and properly. An observed problem has been that students base their mental models completely or partially on intuition and incorrect assumptions, instead of on definitions and proved theorems where validity is ensured. This may be a bad habit stemming from the way math and physics is taught in earlier stages of the education, and we think that DSLs will inherently force students into the right mindset for learning this subject well! Further, many textbooks on physics are incredibly thick and boring, so we've decided to emulate the great and good [Learn You a Haskell for Great Good](#) in order to provide som comic relief in between all the dramatic definitions.

In this book, we will study physics through the lens of DSLs. We will need to pay close attention to definitions, thoughtfully ponder any non-trivial syntax, and verify (via tests or

proofs) that the things we do are actually correct. The areas covered include such things as: dimensional analysis, vectors, calculus, and more!

The book is aimed at you who have some knowledge of [Haskell](#). If you know what a `class` and an `instance` is, you're probably good to go! Even if you don't we're sure you could pick it up as we go. We believe in you!

If you wonder about the weird code blocks at the start of many chapters,

```
module Introduction.About where
```

they are there to declare the chapters as Haskell modules. All chapters are written in *Literate Haskell*, and can be used with GHC/GHCi directly as source code. Therefore, you may choose to read each chapter as documented source code, rather than text with examples.

[src: [Introduction/About.lhs](#)] Previous: [Table of contents](#) [Table of contents](#) Next: [So what's a DSL?](#)

© Björn Werner, Erik Sjöström, Johan Johansson, Oskar Lundström (2018), GPL

# Learn You a **Physics** for Great Good!

>>> WORK IN PROGRESS <<<

## Introduction / So what's a DSL?

[src: [Introduction/WhatIsADsl.lhs](#)] Previous: [About this book](#) [Table of contents](#) Next: [Getting started](#)

`module Introduction.WhatIsADsl where`

## So what's a DSL?

In general, a domain specific language is simply a computer language for a specific domain. It's NOT a synonym for [jargon](#)! DSLs can be specialized for markup, like [HTML](#); for modelling, like [EBNF](#); for shell scripting, like [Bash](#); and more.

The languages we will construct will mostly concern modelling of physics and mathematics. We will create data structures in Haskell to represent the same physics calculations that we write on paper, in such a way that we can write functions to, for example, analyze the expressions for validity.

Look, we'll demonstrate. Let's say we want to model a language that is a subset to the common algebra we're all familiar with. Our language will consist expressions of a single variable and addition. For example, the following three expressions are all valid in such a language:

$$x + x$$

$$x$$

$$(x + x) + (x + x)$$

When implementing a DSL, we typically start with modelling the syntax. Let's first declare a data type for the language

```
data Expr
```

Then we interpret the textual description of the language. "Our language will consist of expressions of a single variable and addition". Ok, so an expression can be one of two things then: a single variable

```
= X
```

or two expressions added together.

```
| Add Expr Expr
```

And that's it, kind of! However, a DSL without any associated functions for validation, symbolic manipulation, evaluation, or somesuch, is really no DSL at all! We must DO something with it, or there is no point!

One thing we can do with expressions such as these, is compare whether two of them are equal. Even without using any numbers, we can test this by simply counting the *x*s! If both expressions contain the same number of *x*s, they will be equal!

```
eq :: Expr -> Expr -> Bool
eq e1 e2 = count e1 == count e2
  where count X           = 1
        count (Add e1 e2) = count e1 + count e2
```

We can now test whether our expressions are equal.

```
ghci> eq (Add (Add X X) X) (Add X (Add X X))
True
```

And NOW that's it (if we want to stop here)! This is a completely valid (but boring) DSL. We've modelled the syntax, and added a function that operates symbolically on our language. This is a very small and simple DSL, and you've likely done something similar before without even realizing you were constructing a DSL. It can really be that simple

In other DSLs we may also look at evaluation and the semantics of a language, i.e. what is the actual type of the result when we *evaluate* or "compute" our expressions.

Throughout this book we'll construct and use DSLs in many different ways. Different parts of the physics we look at will require different DSL treatments, basically. There is no *one* model to use for all DSLs.

[src: [Introduction/WhatsADsl.lhs](#)] Previous: [About this book](#) [Table of contents](#) Next: [Getting started](#)

© Björn Werner, Erik Sjöström, Johan Johansson, Oskar Lundström (2018), GPL



# Learn You a **Physics** for Great Good!

>>> WORK IN PROGRESS <<<

## Introduction / Getting started

[src: [Introduction/GettingStarted.lhs](#)]Previous: [So what's a DSL?Table of contents](#)Next: [Introduction](#)

`module Introduction.GettingStarted where`

## What you need to dive in

To just follow along and implement the same stuff we do, a single document loaded into GHCi will do. For this, you just need to install the [Haskell Platform](#).

If you want to automatically install any required dependencies, like [Hatlab](#) which will be used later on, install [Stack](#). Stack is a build system that will automatically get dependencies and build the project for you. If you want to build the code the same way we do, this is what you'll need. With Stack installed and [our repository cloned](#), enter the Physics directory and type `stack build`. Stack should now download and compile all necessary dependencies, such that they are available when you load a module in GHCi.

## If you need to sharpen up your skills

If you need to freshen up on your Haskell, [Learn You a Haskell for Great Good](#) is a great book that covers all the important stuff in a humorous manner that really holds your attention.

If you still don't really get what DSLs are all about, try reading the ["notes" \(almost a full book really\) for the course on DSLs of math at Chalmers](#). If you're studying at Chalmers, it is even better to actually take the course!

If there's some part of the material that you still think is unclear (or maybe even wrong? Blasphemy!), please [open an issue on the github repo!](#). We'll look into it!

[src: [Introduction/GettingStarted.lhs](#)]Previous: [So what's a DSL?Table of contents](#)Next: [Introduction](#)

© Björn Werner, Erik Sjöström, Johan Johansson, Oskar Lundström (2018), GPL

# Learn You a **Physics** for Great Good!

>>> **WORK IN PROGRESS** <<<

## Calculus / Introduction

[src: [Calculus/Intro.lhs](#)] Previous: [Getting started](#) [Table of contents](#) Next: [Function expressions](#)

module Calculus.Intro where

## What is calculus?

Plain equations where no values change over time are all fine and well. The importance of being able to solve basic problems like “If Jenny has 22 apples, and Richard has 18 apples: how many apples do they have together?” cannot be understated, but they’re not especially fun!

“An unstoppable car has a constant velocity of 141.622272 km/h. How many kilometers has it driven after a day?”. To solve more interesting problems like this, we need calculus.

Calculus is the study of stuff that continuously change over time (or some other continuous variable). For example, a distance that changes over time is equivalent to a velocity. You can have rates of changes with respect to other units, like length, as well, but those are not as common.

There are two major branches of calculus, differential calculus and integral calculus. Differential calculus is all about those rates of changes and graph slopes. Differences, differentials, derivatives, and the like. Integral calculus, on the other hand, is all about accumulation and areas. Sums, integrals, and such.

In this chapter we'll explore the syntax of differences, the problem with differentials, symbolic differentiation, and numeric and symbolic integration.

[src: [Calculus/Intro.lhs](#)] Previous: [Getting started](#) [Table of contents](#) Next: [Function expressions](#)

© Björn Werner, Erik Sjöström, Johan Johansson, Oskar Lundström (2018), GPL

# Learn You a **Physics** for Great Good!

## >>> WORK IN PROGRESS <<<

### Calculus / Function expressions

[src: [Calculus/FunExpr.lhs](#)] Previous: [Introduction](#) [Table of contents](#) Next: [Differential calculus](#)

```
module Calculus.FunExpr (FunExpr (..), RealNum, RealFunc) where
```

```
import Test.QuickCheck
```

## Semantics, syntax, and stuff

What is a value in calculus? What kind of values do functions in calculus operate on and produce?

Let's look at derivatives to get an idea of what the semantic value of calculus is.

$$\frac{dx^2}{dx} = 2x$$

$$\frac{df(x)}{dx} = f'(x)$$

Hmm, these examples left me more confused than before. The differentiation function seems to take an expression as an argument, and return the differentiated expression, with regards to a variable. But what is an expression represented as a semantic value? It's not a number yet, the variable in the body needs to be substituted first in order for the expression to be computable. Is it some kind of function then? Well, yes it is! If we reinterpret the differentiation expressions above, it makes more sense.

$$\frac{dx^2}{dx} = 2x$$

can be written as

$$D(x^2) = 2x \text{ with regards to } x$$

which is really equivalent to

$$D(x \mapsto x^2) = x \mapsto 2x$$

or with more descriptive function names

$$D(\text{square}) = \text{double}$$

So the type of unary real functions seems like a great fit for a semantic value for calculus, and it is! Great! But... how do we represent a real number in Haskell? There is no `Real` type to use. Well, for the sake of simplicity we can just say that a real number is basically the same as a `Double`, and really it is (basically). The problem with `Double` is that it's of finite precision, so rounding errors may occur. We'll have to keep that in mind when doing calculations!

```
type RealNum = Double
type RealFunc = RealNum -> RealNum
```

Now, to the syntax. We've concluded that real functions are really what calculus is all about, so let's model them. We create a data type `FunExpr` that will represent symbolic expressions of functions in our language.



`FunExpr` is very expressive!

```
data FunExpr
```

First of all, there are the elementary functions. We can't have them all, that would get too repetitive to implement, but we'll put in all the fun ones.

```
= Exp | Log | Sin | Cos | Asin | Acos
```

Then, there are the arithmetic operators. "But wait", you say, "Aren't arithmetic operators used to combine expressions, not functions?". We hear you friend, but we will do it anyways. We could make a Lambda constructor for "VAR  $\mapsto$  EXPR" expressions and define the arithmetic operators for the expression type, but this would make our language much more complicated! Instead, we'll restrain ourselves to single variable expressions, which can be represented as compositions of unary functions, and define the arithmetic operators for the functions instead.

$$f \text{ OP}_{r \rightarrow r} g = x \mapsto (f(x) \text{ OP}_r g(x))$$

```
| FunExpr :+ FunExpr
| FunExpr :- FunExpr
| FunExpr :* FunExpr
| FunExpr :/ FunExpr
| FunExpr :^ FunExpr
```

And then there's that single variable. As everything is a function expression, the function that best represents "just a variable" would be  $x \mapsto x$ , which is the same as the *id* function.

```
| Id
```

In a similar vein, the constant function.  $\text{const}(c) = x \mapsto c$

```
| Const RealNum
```

Then there's function composition. If you didn't already know it, it's defined as

$$f \circ g = x \mapsto f(g(x))$$

```
| FunExpr :. FunExpr
```

Finally, the real heroes: the functions of difference, differentiation, and integration! They will be well explored later. But for now, we just define the syntax for them as

```
| Delta RealNum FunExpr
| D FunExpr
| I FunExpr
```

Even more finally, we add a deriving modifier to automatically allow for syntactic equality tests between FunExprs.

```
deriving Eq
```

Nice! This syntax tree will allow us to do symbolically (at the syntax level) what we otherwise would have to do numerically (at the semantics level).

Before we move on, we just have to fix one thing: the operator precedence! If we don't do anything about it, this will happen

```
ghci> Id :+: Id :* Id == (Id :+: Id) :* Id
True
```

Now this is obviously wrong. *Plus* doesn't come before *times*, unless I accidentally switched timelines in my sleep. To fix this, we have to fix the fixity. `infixl` allows us to make an operator left-associative, and set the precedence.

```
-- Medium precedence
infixl 6 :+:
infixl 6 :-
-- High precedence
infixl 7 :*
infixl 7 :/
-- Higher precedence
infixl 8 :^
-- Can't go higher than this precedence
infixl 9 :.
```

## A structure with class

Now that we've defined the basic structure of our language, we can instantiate some useful classes. There are three in particular we care for: `Show`, `Num`, and `Arbitrary`.

Try modifying `FunExpr` to derive `Show`, so that our expressions can be printed.

```
deriving (Eq, Show)
```

Consider now how GHCi prints out a function expression we create



```

ghci> carAccel = Const 20
ghci> carSpeed = Const 50 :+: carAccel :* Id
ghci> carPosition = Const 10 :+: carSpeed :* Id
ghci> carPosition
Const 10.0 :+: (Const 50.0 :+: Const 20.0 :* Id) :* Id

```

Well that's borderline unreadable. Further, the grokability of a printed expression is very inversely proportional to the size/complexity of the expression, as I'm sure you can imagine.

So if the derived show is bad, we'll just have to make our own show!

```

showFe :: FunExpr -> String
showFe Exp = "exp"
showFe Log = "log"
showFe Sin = "sin"
showFe Cos = "cos"
showFe Asin = "asin"
showFe Acos = "acos"
showFe (f :+: g) = "(" ++ showFe f ++ " + " ++ showFe g ++ ")"
showFe (f :- g) = "(" ++ showFe f ++ " - " ++ showFe g ++ ")"
showFe (f :* g) = "(" ++ showFe f ++ " * " ++ showFe g ++ ")"
showFe (f :/ g) = "(" ++ showFe f ++ " / " ++ showFe g ++ ")"
showFe (f :^ g) = "(" ++ showFe f ++ "^" ++ showFe g ++ ")"
showFe Id = "id"
showFe (Const x) = showReal x
showFe (f :. g) = "(" ++ showFe f ++ " . " ++ showFe g ++ ")"
showFe (Delta h f) = "(delta_" ++ showReal h ++ " " ++ showFe f ++ ")"
showFe (D f) = "(D " ++ showFe f ++ ")"
showFe (I f) = "(I " ++ showFe f ++ ")"

showReal x = if isInt x then show (round x) else show x
  where isInt x = x == fromInteger (round x)

instance Show FunExpr where
  show = showFe

```

Not much to explain here. It's just one way to print our syntax tree in a more readable way. What's interesting is how we can now print our expressions in a much more human friendly way!

```

ghci> carPosition
(10 + ((50 + (20 * id)) * id))

```

Still a bit noisy with all the parentheses, but much better!

Another class we can instantiate for our `FunExpr` is `Num`. This class allows us to make use of the native Haskell functions and operators for numeric operations, instead of writing our own constructors. This sometimes improves the readability of the code.

```
instance Num FunExpr where
  negate e    = Const 0 :- e
  (+)         = (:+)
  (*)         = (:*)
  fromInteger = Const . fromInteger
  abs         = undefined
  signum      = undefined
```

Third but not least, we'll instantiate `Arbitrary`. This class is associated with the testing library *QuickCheck*, and describes how to generate arbitrary values of a type for use when testing logical properties with `quickCheck`. For example, a property function could be formulated that states that the  `:*`  constructor of `FunExpr` is associative.

The implementation itself is not very interesting. We generate a function expression that tends to contain mostly elementary functions, arithmetic operations, and a generous dose of constants; with a light sprinkle of differences. We won't include derivatives as all elementary functions have elementary derivatives anyways, and integrals may cause cause approximation errors if we have to numerically compute them at evaluation.

```
instance Arbitrary FunExpr where
  arbitrary =
```

frequency "chooses one of the given generators, with a weighted random distribution". By assigning probabilities of generating certain functions more often than others, we can restrain the growth of the generated expressions in complexity.

```
frequency
  [ (10, genElementary)
  , (10, genBinaryOperation)
  , (10, return Id)
  , (20, fmap Const arbitrary)
  , (10, genBinaryApp (:.))
  , (5 , genBinaryApp Delta) ]
where genElementary = elements [Exp, Log, Sin, Cos, Asin, Acos]
```

```
genBinaryApp op = fmap (\(f, g) -> f `op` g) arbitrary
genBinaryOperation =      elements [(:+), (: -), (: *), (: /), (: ^)]
                        >=> genBinaryApp
```

[src: [Calculus/FunExpr.lhs](#)] Previous: [Introduction](#) [Table of contents](#) Next: [Differential calculus](#)

© Björn Werner, Erik Sjöström, Johan Johansson, Oskar Lundström (2018), GPL

# Learn You a **Physics** for Great Good!

>>> WORK IN PROGRESS <<<

## Calculus / Differential calculus

[src: [Calculus/DifferentialCalc.lhs](#)]

Previous: [Function expressions](#)

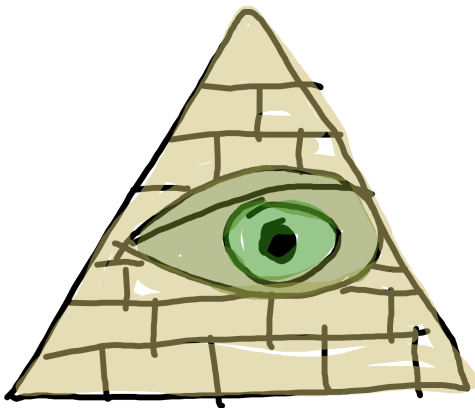
[Table of contents](#)

Next: [Integral calculus](#)

```
module Calculus.DifferentialCalc (deriveApprox, derive, simplify) where

import Calculus.FunExpr
```

## Deep, dark, differences



A *difference* is, in it's essence, quite simply the result of applying the operation of subtraction to two real number terms.

$$\textit{minuend} - \textit{subtrahend} = \textit{difference}$$

Nice, great job, we're done here, let's move on.

...

Just kidding, of course there's more to it than that.

In calculus, the term *difference* carries more meaning than usual. More than just a subtraction of arbitrary values, differences lie at the heart of calculations regarding rate of change, both average and instantaneous.

Quotients of differences of functions of the same time describe the average rate of change over the time period. For example, an average velocity can be described as the difference quotient of difference in position and difference in time.

$$v_{avg} = \frac{p_2 - p_1}{t_2 - t_1}$$

where  $p_n$  is the position at time  $t_n$ .

In the context of calculus, we use a special syntax for differences: the delta operator! With this, the previous definition can be rewritten as

$$v_{avg} = \frac{p_2 - p_1}{t_2 - t_1} = \frac{\Delta p}{\Delta t}$$

This is the informal definition of the delta operator used in *University Physics*:

$$\Delta x = x_2 - x_1$$

Ok, so it's a difference. But what does  $x_2$  and  $x_1$  mean, and what do they come from?  $x_2$  and  $x_1$  are not explicitly bound anywhere, but it seems reasonable to assume that  $x_i \in \mathbb{R}$  or equivalently, that  $x$  is a function with a subscript index as an argument, that returns a  $\mathbb{R}$ .

Further, the indices 1, 2 should not be thought of as specific constants, but rather arbitrary real number variables identified by these integers. Lets call them  $a, b$  instead, to make it clear that they are not constants.

$$\Delta x = x_b - x_a$$

Now  $a, b$  are implicitly bound. We make the binding explicit.

$$(\Delta x)(a, b) = x_b - x_a$$

We compare this to the more formal definition of *forward difference* on wikipedia:

$$\Delta_h[f](x) = f(x + h) - f(x)$$

The parameter bindings are a bit all over the place here. To more easily compare to our definition, let's rename  $x$  to  $a$  and  $f$  to  $x$ , and change the parameter declaration syntax:

$$(\Delta x)(h)(a) = x(a + h) - x(a)$$

This is almost identical to the definition we arrived at earlier, with the exception of expressing  $b$  as  $a + h$ . Why is this? Well, in calculus we mainly use differences to express two things, as mentioned previously. Average rate of change and instantaneous rate of change.

Average rate of change is best described as the difference quotient of the difference in y-axis value over an interval of  $x$ , and the difference in x-axis value over the same interval.

$$\frac{y(x_b) - y(x_a)}{x_b - x_a}$$

In this case, the  $x$ 's can be at arbitrary points on the axis, as long as  $b > a$ . Therefore, the definition of difference as  $(\Delta x)(a, b) = x_b - x_a$  seems a good fit. Applied to average velocity, our difference quotient

$$v_{avg} = \frac{\Delta p}{\Delta t}$$

will expand to

$$v_{avg}(t_2, t_1) = \frac{(\Delta p)(t_2, t_1)}{(\Delta t)(t_2, t_1)}$$

for  $t_2 > t_1$ .

Instantaneous rate of change is more complicated. At its heart, it too is defined in terms of differences. However, we are no longer looking at the average change over an interval delimited by two points, but rather the instantaneous change in a single point.

Of course, you can't have a difference with only one point. You need two points to look at how the function value changes between them. But what if we make the second point reeeeeeeeeealy close to the first? That's basically the same as the difference in a single point, for all intents and purposes. And so, for instantaneous rate of change, the definition of difference as  $(\Delta x)(h)(a) = x(a + h) - x(a)$  will make more sense, for very small  $h$ 's. Applied to instantaneous velocity, our difference quotient

$$v_{inst} = \frac{\Delta p}{\Delta t}$$

for very small  $\Delta t$ , will expand to

$$v_{inst}(h, x) = \frac{(\Delta p)(h, x)}{(\Delta t)(h, x)}$$

for very small  $h$ .

As  $h$  gets closer to 0, our approximation of instantaneous rate of change gets better.

And so, we have a method of computing average rate of change, and instantaneous rate of change (numerically approximatively). In Haskell, we can make shallow embeddings for differences in the context of rate of change as velocity.

Average velocity is simply

```
v_avg pos t2 t1 = (pos(t2) - pos(t1)) / (t2 - t1)
```

which can be used as

```
ghci> v_avg (\x -> 5*x) 10 0
5.0
```

And instantaneous velocity is

```
v_inst pos h t = (pos(t + h) - pos(t)) / ((t + h) - t)
```

which can be used as

```
ghci> carSpeed t = v_inst (\x -> x + sin(x)) 0.00001 t
ghci> carSpeedAtPointsInTime = map carSpeed [0, 1, 2, 3]
ghci> carSpeedAtPointsInTime
[1.9999999999833333,1.5402980985023251,0.5838486169602084,1.0006797790330592e-2]
```

We'd also like to model one of the versions of the delta operator, finite difference, in our syntax tree. As the semantic value of our calculus language is the unary real function, the difference used for averages doesn't really fit in well, as it's a binary function (two arguments:  $t_2$  and  $t_1$ ). Instead, we'll use the version of delta used for instants, as it only takes a single point in time as an argument (assuming  $h$  is already given).

The constructor in our syntax tree is therefore

| Delta RealNum FunExpr

where the first argument is  $h$ , and the second is the function.

## Derivatives

The *derivative* of a function is, according to wikipedia, “the slope of the tangent line to the graph of [a] function at [a] point” and can be described as the “instantaneous rate of change”, and *differentiation* is the method of finding a derivative for a function.

...

Wait, didn't we just look at instantaneous rates of changes in the previous section on differences? Well yes, and the difference quotient for a function at a point with a very small step  $h$  is indeed a good way to numerically approximate the derivative of a function. From what we found then, we can derive a general expression for instantaneous rate of change

$$\frac{(\Delta f)(h, x)}{(\Delta id)(h, x)} = \frac{f(x + h) - f(x)}{h}$$

for very small  $h$ .

But what if we don't want just a numerical approximation, but THE derivative of a function at any arbitrary point? What if we make  $h$  not just very small, but *infinitely* small?

Introducing *infinitesimals*! From the wikipedia entry on *Leibniz's notation*

In calculus, Leibniz's notation, named in honor of the 17th-century German philosopher and mathematician Gottfried Wilhelm Leibniz, uses the symbols  $dx$  and  $dy$  to represent infinitely small (or infinitesimal) increments of  $x$  and  $y$ , respectively, just as  $\Delta x$  and  $\Delta y$  represent finite increments of  $x$  and  $y$ , respectively.

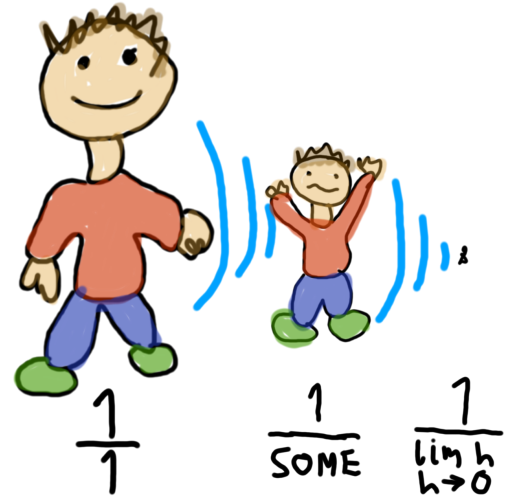


So there's a special syntax for differences where the step  $h$  is infinitely small, and it's called Leibniz's notation. We could naively interpret the above quote in mathematical terms as

$$dx = \lim_{\Delta x \rightarrow 0} \Delta x$$

such that

$$\forall y(x). D(y) = \frac{dy}{dx} = \frac{\lim_{\Delta y \rightarrow 0} \Delta y}{\lim_{\Delta x \rightarrow 0} \Delta x}$$



where  $D$  is the function of differentiation. However, this doesn't quite make sense as  $\lim_{x \rightarrow 0} x = 0$  and we'd be dividing by 0. The wording here is important: infinitesimals aren't 0, but *infinitely* small! The result is that we can't define infinitesimals in terms of limits, but have to treat them as an orthogonal concept.

Except for this minor road bump, this definition of derivatives is very appealing, as it suggests a very simple and intuitive transition from finite differences to infinitesimal differentials. Also, it suggests the possibility of manipulating the infinitesimals of the derivative algebraically, which might be very useful. However, this concept is generally considered too imprecise to be used as the foundation of calculus.

A later section on the same wikipedia entry elaborates a bit:

Leibniz's concept of infinitesimals, long considered to be too imprecise to be used as a foundation of calculus, was eventually replaced by rigorous concepts developed by Weierstrass and others. Consequently, Leibniz's quotient notation was re-interpreted to stand for the limit of the modern definition. However, in many instances, the symbol did seem to act as an actual quotient would and its usefulness kept it popular even in the face of several competing notations.

What is then the "right" way to do derivatives? As luck would have it, not much differently than Leibniz's suggested and how we interpreted it above! The intuitive idea can be turned into a precise definition by defining the derivative to be the limit of difference quotients of real numbers. Again, from wikipedia - Leibniz's notation:

In its modern interpretation, the expression  $dy/dx$  should not be read as the division of two quantities  $dx$  and  $dy$  (as Leibniz had envisioned it); rather, the

whole expression should be seen as a single symbol that is shorthand for

$$D(y) = \lim_{\Delta x \rightarrow 0} \frac{\Delta y}{\Delta x}$$

which, when  $y$  is a function of  $x$ , and  $x$  is the *id* function for real numbers (which it is in the case of time), is:

$$\begin{aligned} D(y) &= \lim_{\Delta x \rightarrow 0} \frac{\Delta y}{\Delta x} \\ &= a \mapsto \lim_{\Delta x \rightarrow 0} \frac{(\Delta y)(\Delta x, a)}{\Delta x} \\ &= a \mapsto \lim_{h \rightarrow 0} \frac{y(a + (\Delta x)(h, a)) - y(a)}{(\Delta x)(h, a)} \\ &= a \mapsto \lim_{h \rightarrow 0} \frac{y(a + ((a + h) - a)) - y(a)}{(a + h) - a} \\ &= a \mapsto \lim_{h \rightarrow 0} \frac{y(a + h) - y(a)}{h} \end{aligned}$$

There, the definition of derivatives! Not too complicated, was it? We can write a simple numerically approximative according to the definition like

```
deriveApprox f h x = (f(x + h) - f(x)) / h
```

Only when  $h$  is infinitely small is `deriveApprox` fully accurate. However, as we can't really represent an infinitely small number in finite memory, the result will only be approximative, and the approximation will (in most cases) get better as  $h$  gets smaller. For example, let's calculate the slope of  $f(x) = x^2$  at  $x = 3$ . As the slope of this parabola is calculated as  $k = 2x$ , we expect the result of `deriveApprox` to approach  $k = 2x = 2 * 3 = 6$  as  $h$  gets smaller

```
ghci> deriveApprox (\x -> x^2) 5 3
11
ghci> deriveApprox (\x -> x^2) 1 3
7
ghci> deriveApprox (\x -> x^2) 0.5 3
6.5
ghci> deriveApprox (\x -> x^2) 0.1 3
6.1000000000000012
ghci> deriveApprox (\x -> x^2) 0.01 3
6.0099999999999849
```

Ok, that works, but not well. By making use of that fancy definition for derivatives that we derived earlier, we can now derive things symbolically, which implies provable 100% perfect accuracy, no numeric approximations!

We define a function to symbolically derive a function expression. `derive` takes a function expression, and returns the differentiated function expression.

```
derive :: FunExpr -> FunExpr
```

Using only the definition of derivatives, we can derive the definitions of `derive` for the various constructors in our syntax tree.

For example, how do we derive `f :+: g`? Let's start by doing it mathematically.

$$\begin{aligned}
 D(f + g) &= a \mapsto \lim_{h \rightarrow 0} \frac{(f + g)[a + h] - (f + g)[a]}{h} \\
 &\quad \{ \text{Addition of functions} \} \\
 &= a \mapsto \lim_{h \rightarrow 0} \frac{f(a + h) + g(a + h) - (f(a) + g(a))}{h} \\
 &= a \mapsto \lim_{h \rightarrow 0} \frac{f(a + h) + g(a + h) - f(a) - g(a)}{h} \\
 &= a \mapsto \lim_{h \rightarrow 0} \left( \frac{f(a + h) - f(a)}{h} + \frac{g(a + h) - g(a)}{h} \right) \\
 &= a \mapsto \left( \lim_{h \rightarrow 0} \frac{f(a + h) - f(a)}{h} \right) + \left( \lim_{h \rightarrow 0} \frac{g(a + h) - g(a)}{h} \right) \\
 &= \left( a \mapsto \lim_{h \rightarrow 0} \frac{f(a + h) - f(a)}{h} \right) + \left( a \mapsto \lim_{h \rightarrow 0} \frac{g(a + h) - g(a)}{h} \right) \\
 &\quad \{ \text{Definition of derivative} \} \\
 &= D(f) + D(g)
 \end{aligned}$$

Oh, it's just the sum of the derivatives of both functions! The Haskell implementation is then trivially

```
derive (f :+: g) = derive f :+: derive g
```

Let's do one more, say, *sin*. We will make use of the trigonometric identity of sum-to-product

$$\sin \theta - \sin \varphi = 2 \sin \left( \frac{\theta - \varphi}{2} \right) \cos \left( \frac{\theta + \varphi}{2} \right)$$

And the limit

$$\lim_{x \rightarrow 0} \frac{\sin x}{x} = 1$$

the proof of which is left as an exercise to the reader

**Exercise.** Prove the limit  $\lim_{x \rightarrow 0} \frac{\sin x}{x} = 1$

▼ Hint

This limit can be proven using the [unit circle](#) and [squeeze theorem](#)

Then, the differentiation

$$\begin{aligned} D(\sin) = a \mapsto \lim_{h \rightarrow 0} \frac{\sin(a+h) - \sin(a)}{h} \\ & \{ \text{trig. sum-to-product} \} \\ & = a \mapsto \lim_{h \rightarrow 0} \frac{2 \sin \left( \frac{a+h-a}{2} \right) \cos \left( \frac{a+h+a}{2} \right)}{h} \\ & = a \mapsto \lim_{h \rightarrow 0} \frac{2 \sin \left( \frac{h}{2} \right) \cos \left( \frac{2a+h}{2} \right)}{h} \\ & = a \mapsto \lim_{h \rightarrow 0} \frac{2 \sin \left( \frac{h}{2} \right) \cos \left( \frac{2a+h}{2} \right)}{h} \\ & = a \mapsto \lim_{h \rightarrow 0} \frac{\sin \left( \frac{h}{2} \right)}{\frac{h}{2}} \cos \left( \frac{2a+h}{2} \right) \\ & \{ h \text{ approaches } 0 \} \\ & = a \mapsto 1 \cos \left( \frac{2a+0}{2} \right) \\ & = a \mapsto \cos(a) \\ & = \cos \end{aligned}$$

Again, trivial definition in Haskell

```
derive Sin = Cos
```

**Exercise.** Derive the rest of the cases using the definition of the derivative

### ▼ Solution

```
derive Exp = Exp
derive Log = Const 1 :/ Id
derive Cos = Const 0 :- Sin
derive Asin = Const 1 :/ (Const 1 :- Id:^(Const 2)):^(Const 0.5)
derive Acos = Const 0 :- derive Asin
derive (f :- g) = derive f :- derive g
derive (f :* g) = derive f :* g :+ f :* derive g
derive (f :/ g) = (derive f :* g :- f :* derive g) :/ (g:^(Const 2))
derive (f :^ g) = f:^(g :- Const 1)
                  :* (g :* derive f :+ f :* (Log :. f) :* derive g)
derive Id = Const 1
derive (Const _) = Const 0
```

For a function composition  $f \circ g$  we have to handle the case where  $f$  is a constant function, as we may get division by zero if we naively apply the chain rule. We'll make use of the `simplify` function, which we'll define later, to reduce compositions of constant functions to just constant functions.

```
derive (f :. g) =
  case simplify f of
    Const a -> Const 0
    sf      -> (derive sf :. g) :* derive g
derive (Delta h f) = Delta h (derive f)
derive (D f) = derive (derive f)
```

Oh right, I almost forgot: Integrals. How are you supposed to know how to derive these bad boys when we haven't even covered them yet! We'll prove why this works later, but for now, just know that another name for integral is *Antiderivative*...

```
derive (I f) = f
```

# Keep it simple

So we've got our differentiation function, great! Let's try it out by finding the derivative for a simple function, like  $f(x) = \sin(x) + x^2$ , which should be  $f'(x) = \cos(x) + 2x$ :

```
ghci> f = Sin :+: Id:^(Const 2)
ghci> derive f
(cos + ((id^(2 - 1)) * ((2 * 1) + ((id * (log . id)) * 0))))
```

Oh... that's not very readable. If we simplify it manually we get that the result is indeed as expected

```
(cos + ((id^(2 - 1)) * ((2 * 1) + ((id * (log . id)) * 0))))
cos + (id^1 * (2 + (id * (log . id) * 0)))
cos + (id * (2 + 0))
cos + 2*id
```

But still, we shouldn't have to do that manually! Let's have Mr. Computer help us out, by writing a function to simplify expressions.

We'll write a `simplify` function which will reduce an expression to a simpler, equivalent expression. Sounds good, only... what exactly does "simpler" mean? Is 10 simpler than  $2 + 2 * 4$ ? Well, yes obviously, but there are other expressions where this is not the case. For example, polynomials have two standard forms. The first is the sum of terms, which is appropriate when you want to add or subtract polynomials. The other standard form is the product of irreducible factors, which is a good fit for when you want to divide polynomials.

So, our `simplify` function will not guarantee that every expression is reduced to *its most simple* form, but rather that many expressions will be reduced to *a simpler form*. As an exercise, you can implement more reduction rules to make expressions simpler to you. For example, the trigonometric identities like  $\sin(\theta + \frac{\pi}{2}) = \cos(\theta)$ .

```
simplify :: FunExpr -> FunExpr
```

The elementary functions by themselves are already as simple as can be, so we don't have to simplify those. When it comes to the arithmetic operations, most interesting is the cases of one operand being the identity element.

For addition and subtraction, it's 0

```

simplify (f :+: g) = case (simplify f, simplify g) of
  (Const 0, g') -> g'
  (f', Const 0) -> f'
  (Const a, Const b) -> Const (a + b)
  (f', g') | f' == g' -> simplify (Const 2 :* f')
  (Const a :* f', g') | f' == g' -> simplify (Const (a + 1) :* f')
  (f', Const a :* g') | f' == g' -> simplify (Const (a + 1) :* f')
  (Const a :* f', Const b :* g') | f' == g'
    -> simplify (Const (a + b) :* f')
  (f', g') -> f' :+: g'
simplify (f :- g) = case (simplify f, simplify g) of
  (f', Const 0 :- g') -> simplify (f' :+: g')
  (f', Const 0) -> f'
  (Const a, Const b) -> if a > b
    then Const (a - b)
    else Const 0 :- Const (b - a)
  (f', g') | f' == g' -> Const 0
  (Const a :* f', g') | f' == g' -> simplify (Const (a - 1) :* f')
  (f', Const a :* g') | f' == g'
    -> Const 0 :- simplify (Const (a - 1) :* f')
  (Const a :* f', Const b :* g') | f' == g'
    -> simplify ((Const a :- Const b) :* f')
  (f', g') -> f' :- g'

```

For multiplication and division, the identity element is 1, but the case of one operand being 0 is also interesting

```

simplify (f :* g) = case (simplify f, simplify g) of
  (Const 0, g') -> Const 0
  (f', Const 0) -> Const 0
  (Const 1, g') -> g'
  (f', Const 1) -> f'
  (Const a, Const b) -> Const (a * b)
  (f', Const c) -> Const c :* f'
  (f', g') | f' == g' -> f' :^ Const 2
  (Const a, Const b :* g') -> simplify (Const (a*b) :* g')
  (Const a :* f', g') -> simplify (f' :* (Const a :* g'))
  (fa, g' :/ fb) | fa == fb -> g'
  (f', g') -> f' :* g'
simplify (f :/ g) = case (simplify f, simplify g) of
  (Const 0, g') -> Const 0
  (f', Const 1) -> f'
  (f', g') | f' == g' -> Const 1
  (f', g') -> f' :/ g'

```

Exponentiation is not commutative, and further has no (two-sided) identity element. However, it does have an “asymmetric” identity element: the right identity 1!

```
simplify (f :^ g) = case (simplify f, simplify g) of
  (f', Const 1) -> f'
  (f', g') -> f' :^ g'
```

**Exercises.** Look up (or prove by yourself) more identities (of expressions, not identity elements) for exponentiation and implement them.

### ▼ Solution

For example, there is the identity of negative exponents. For any integer  $n$  and nonzero  $b$

$$b^{-n} = \frac{1}{b^n}$$

Intuitively, the identity function is the identity element for function composition

```
simplify (f :: g) = case (simplify f, simplify g) of
  (Const a, _) -> Const a
  (Id, g') -> g'
  (f', Id) -> f'
  (f', g') -> f' :: g'
```

```
simplify (Delta h f) = Delta h (simplify f)
simplify (D (I f')) = simplify f'
simplify (D f) = D (simplify f)
simplify (I f) = I (simplify f)
simplify f = f
```

With this new function, many expressions become much more readable!

```
ghci> f = Sin :+: Id:^(Const 2)
ghci> derive f
(cos + ((id^(2 - 1)) * ((2 * 1) + ((id * (log . id)) * 0))))
ghci> simplify (derive f)
(cos + (2 * id))
```



A sight for sore eyes!

**Exercise.** Think of more ways an expression can be “simplified”, and add your cases to the implementation.

[src:  
[Calculus/DifferentialCalc.lhs](#)]

Previous: [Function  
expressions](#)

[Table of  
contents](#)

Next: [Integral  
calculus](#)

© Björn Werner, Erik Sjöström, Johan Johansson, Oskar Lundström (2018), GPL

# Learn You a **Physics** for Great Good!

>>> WORK IN PROGRESS <<<

## Calculus / Integral calculus

[src: [Calculus/IntegralCalc.lhs](#)] Previous: [Differential calculus](#) [Table of contents](#) Next: [Plotting graphs](#)

```
module Calculus.IntegralCalc (integrateApprox, integrate, eval) where

import Calculus.FunExpr
import Calculus.DifferentialCalc
```

## Integrals - An integral part of calculus

*Integrals* are functions used to describe area, volume, and accumulation in general. The operation of integration is the second fundamental operation of calculus, and the inverse of differentiation. Whereas derivatives are used to describe the rate of change in an instant, integrals are used to describe the accumulation of value over time.

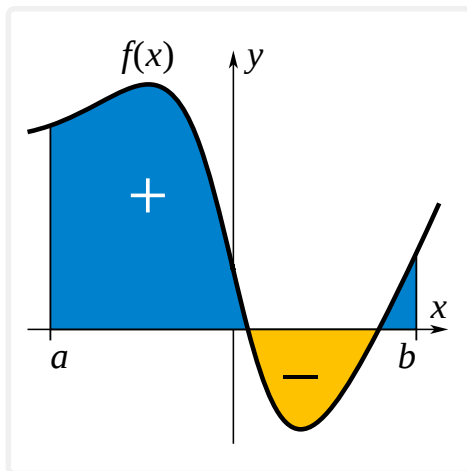
Recall how we used derivatives before. If we know the distance traveled of a car and the time it took, we can use differentiation to calculate the velocity. Similarly but reversely, if we know the velocity of the car and the time it travels for, we can use integration to calculate the distance traveled.

$$x_{traveled} = \int_{t_0}^{t_1} v(t) dt$$

Ok, let's dive into this! We need to grok the syntax and find a rigorous, modelable definition of what *exactly* an integral is. We ask our kind friend Wikipedia for help. From the entry on *Integral*:

Given a function  $f$  of a real variable  $x$  and an interval  $[a, b]$  of the real line, the definite integral

$$\int_a^b f(x)dx$$



A definite integral of a function can be represented as the signed area of the region bounded by its graph. (C) KSmrq

is defined informally as the signed area of the region in the  $xy$ -plane that is bounded by the graph of  $f$ , the  $x$ -axis and the vertical lines  $x = a$  and  $x = b$ . The area above the  $x$ -axis adds to the total and that below the  $x$ -axis subtracts from the total.

Roughly speaking, the operation of integration is the inverse of differentiation. For this reason, the term integral may also refer to the related notion of the antiderivative, a function  $F$  whose derivative is the given function  $f$ . In this case, it is called an indefinite integral and is written:

$$F(x) = \int f(x)dx$$

Ok, so first of all: confusion. Apparently there are two different kinds of integrals, *definite integrals* and *indefinite integrals*?

Let's start with defining *indefinite* integrals. *Wikipedia - Antiderivative* tells us that the *indefinite* integral, also known as the *antiderivative*, of a function  $f$  is equal to a differentiable function  $F$  such that  $D(F) = f$ . It further tells us that the process of finding the antiderivative is called *antidifferentiation* or *indefinite integration*.

The same article then brings further clarification

Antiderivatives are related to definite integrals through the fundamental theorem of calculus: the definite integral of a function over an interval is equal to the difference between the values of an antiderivative evaluated at the endpoints of the interval.

So indefinite integrals are the inverse of derivatives, and definite integrals are just the application of an indefinite integral to an interval. If we look back at the syntax used, this makes sense.

$\int f(x)dx$  is the indefinite integral. A function not applied to anything.  $\int_a^b f(x)dx$  is the definite

integral. The difference of the indefinite integral being applied to the endpoints of the interval  $[a, b]$ .

To simplify a bit, we see that just as with derivatives the  $x$ 's everywhere are just there to confuse us, so we remove them.

$$\int f(x)dx$$

should really just be

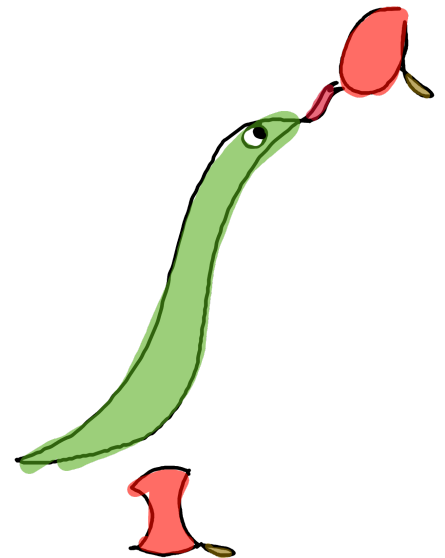
$$\int f$$

Next, the definition of definite integrals implies that we can write

$$\int_a^b f(x)dx$$

as

$$\left(\int f\right)[b] - \left(\int f\right)[a]$$



Only one of the two kinds of integral are fit to directly model in the syntax tree of our language. As `FunExpr` represents functions, it has to be the indefinite integral, which is a function unlike the definite integral which is a real value difference.

A thing to note is that while we may sometimes informally speak of the indefinite integral as a single unary function like any other, it's actually a set of functions, and the meaning of  $F(x) = \int f(x)dx$  is really ambiguous. The reason for this is that for some function  $f$ , there is not just one function  $F$  such that  $D(F) = f$ . A simple counterexample is

$$D(x \mapsto x + 2) = 1 \text{ and } D(x \mapsto x + 3) = 1$$

The fact that adding a constant to a function does not change the the derivative, implies that the indefinite integral of a function is really a set of functions where the constant term differs.

$$\int f = \{F + \text{const } C \mid C \in \mathbb{R}\}$$

We don't want sets though. We want unary real functions (because that's our the type of our semantics!). So, we simply say that when integrating a function, the constant term will always be 0 in order to nail the result down to a single function! I.e.  $(If)0 = 0$

| I FunExpr

## Actually integrating with my man, Riemann

We've analyzed *what* an integral is, and we can tell if a function is the antiderivative of another. For example,  $x^2$  is an antiderivative of  $2x$  because  $D(x^2) = 2x$ . But *how* do we find integrals in the first place?

We start our journey with a familiar name, Leibniz. He, and also but independently Newton, discovered the heart of integrals and derivatives: The *fundamental theorem of calculus*. The definitions they made were all based on infinitesimals which, as said earlier, was considered too imprecise. Later, Riemann rigorously formalized integration using limits.

There exist many formal definitions of integrals, and they're not all equivalent. They each deal with different cases and classes of problems, and some remain in use mostly for pedagogical purposes. The most commonly used definitions are the Riemann integrals and the Lebesgue integrals.

The Riemann integral was the first rigorous definition of the integral, and for many practical applications it can be evaluated by the fundamental theorem of calculus or approximated by numerical integration. However, it is a deficient definition, and is therefore unsuitable for many theoretical purposes. For such purposes, the Lebesgue integral is a better fit.

All that considered, we will use Riemann integrals. While they may be lacking for many purposes, they are probably more familiar to most students (they are to me!), and will be sufficient for the level we're at.

If we look back at the syntax of definite integrals

$$\int_a^b f(x)dx$$

the application of  $f$  and the  $dx$  part actually implies the definition of the Riemann integral. We can read it in english as "For every infinitesimal interval of  $x$ , starting at  $a$  and ending at  $b$ , take the value of  $f$  at that  $x$  (equiv. to taking the value at any point in the infinitesimal interval), and

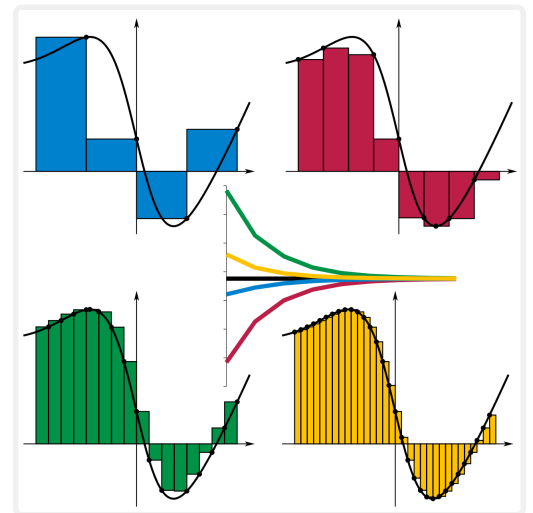
calculate the area of the rectangle with width  $dx$  and height  $f(x)$ , then sum all of these parts together."

As we're dealing with an infinite sum of infinitesimal parts: a limit must be involved.  $a$  and  $b$  are the lower and upper limits of the sum. Our iteration variable should increase with infinitesimal  $dx$  each step. Each step we add the area of the rectangle with height  $f(x')$ , where  $x'$  is any point in  $[x, x + dx]$ . As  $x + dx$  approaches  $x$  when  $dx$  approaches zero,  $x' = \lim_{dx \rightarrow 0} x + dx = x$ .

$$\int_a^b f = \int_a^b f(x)dx = \lim_{dx \rightarrow 0} \sum_{x=a, a+dx, a+2dx, \dots}^b A(x, dx) \text{ where } A(x, dx) = f(x) * dx$$

Smaller  $dx$  result in better approximations. (C) KSmrq

Based on this definition, we could implement a function in Haskell to compute the numerical approximation of the integral by letting  $dx$  be a very small, but finite, number instead of being infinitesimal. The smaller our  $dx$ , the better the approximation



```
integrateApprox :: RealFunc -> RealNum -> RealNum ->
RealNum -> RealNum
integrateApprox f dx a b =
```

$b$  must be greater than  $a$  for a definite integral to make sense, but if that's not the case, we can just flip the order of  $a$  and  $b$  and flip the sign of the area.

```
let area x = f x * dx
in if b >= a
then sum (fmap area (takeWhile (<b) [a + 0*dx, a + 1*dx ..]))
else -sum (fmap area (takeWhile (>b) [a - 0*dx, a - 1*dx ..]))
```

For example, let's calculate the area of the right-angled triangle under  $y = x$  between  $x = 0$  and  $x = 10$ . As the area of a right-angled triangle is calculated as  $A = \frac{b*h}{2}$ , we expect the result of to approach  $A = \frac{b*h}{2} = \frac{10*10}{2} = 50$  as  $dx$  gets smaller

```
λ integrateApprox (\x -> x) 5 0 10
25
λ integrateApprox (\x -> x) 1 0 10
45
λ integrateApprox (\x -> x) 0.5 0 10
47.5
λ integrateApprox (\x -> x) 0.1 0 10
49.500000000000013
```

```
λ integrateApprox (\x -> x) 0.01 0 10
50.049999999999996
```

Great, it works for numeric approximations! This can be useful at times, but not so much in our case. We want closed expressions to use when solving physics problems, regardless of whether there are computations or not!

To find some integrals, making simple use of the fundamental theorem of calculus, i.e.

$D(\int f) = f$ , is enough. That is, we "think backwards". For example, we can use this method to find the integral of  $\cos$ .

Which function derives to  $\cos$ ? Think, think, think ... I got it! It's  $\sin$ , isn't it?

$$D(\sin) = \cos \implies \int \cos = \sin + \text{const}C$$

So simple! The same method can be used to find the integral of polynomials and some other simple functions. Coupled with some integration rules for products and exponents, this can get us quite far! But what if we're not superhumans and haven't memorized all the tables? What if we have to do integration without a cheat sheet for, like, an exam? In situations like these we make use of the definition of the Riemann integral, like we make use of the definition of differentiation in a previous chapter. As an example, let us again integrate  $\cos$ , but now with this second method. Keep in mind that due to the technical limitations of Riemann integrals, not all integrals may be found this way.

Using the trigonometric identity of  $\lim_{x \rightarrow 0} \frac{\sin x}{x} = 1$  we find

$$\begin{aligned}
& \int_a^b \cos \\
& \quad \{ \text{Riemann integral} \} \\
& = \lim_{dx \rightarrow 0} \sum_{x=a, a+dx, a+2*dx, \dots}^b \cos(x) * dx \\
& = \lim_{dx \rightarrow 0} dx * \sum_{x=a, a+dx, a+2*dx, \dots}^b \cos(x) \\
& = \lim_{dx \rightarrow 0} dx * (\cos(a) + \cos(a + dx) + \cos(a + 2 * dx) + \dots + \cos(a + \frac{b-a}{dx} * dx)) \\
& \quad \{ \text{Sums of cosines with arguments in arithmetic progression} \} \\
& = \lim_{dx \rightarrow 0} dx * \frac{\sin(\frac{(\frac{b-a}{dx} + 1)dx}{2}) * \cos(a + \frac{b-a}{dx} * \frac{dx}{2})}{\sin(dx/2)} \\
& = \lim_{dx \rightarrow 0} dx * \frac{\sin(\frac{b-a+dx}{2}) * \cos(\frac{a+b}{2})}{\sin(dx/2)} \\
& \quad \{ \text{Trig. product-to-sum ident.} \} \\
& = \lim_{dx \rightarrow 0} dx * \frac{\sin(\frac{b-a+dx}{2} + \frac{a+b}{2}) + \sin(\frac{b-a+dx}{2} - \frac{a+b}{2})}{2\sin(dx/2)} \\
& = \lim_{dx \rightarrow 0} dx * \frac{\sin(b + dx/2) + \sin(-a + dx/2)}{2\sin(dx/2)} \\
& = \lim_{dx \rightarrow 0} \frac{\sin(b + dx/2) + \sin(-a + dx/2)}{\frac{\sin(dx/2)}{dx/2}} \\
& \quad \{ dx \rightarrow 0 \} \\
& = \frac{\sin(b + 0/2) + \sin(-a + 0/2)}{1} \\
& = \sin(b) + \sin(-a) \\
& = \sin(b) - \sin(a)
\end{aligned}$$

The definition of definite integrals then give us that

$$\int_a^b \cos = \sin(b) - \sin(a) \wedge \int_a^b f = F(b) - F(a) \implies F = \sin$$

The antiderivative of  $\cos$  is  $\sin$  (again, as expected)!

Let's implement these rules as a function for symbolic (indefinite) integration of functions. We'll start with the nicer cases, and progress to the not so nice ones.



integrate takes a function to symbolically integrate, and returns the antiderivative where  $F(0) = 0$ .

Important to note is that not all functions are integrable. Unlike derivatives, some antiderivatives of elementary functions simply cannot be expressed as elementary functions themselves, according to [Liouville's theorem](#). Some examples include  $e^{-x^2}$ ,  $\frac{\sin(x)}{x}$ , and  $x^x$ .

```
integrate :: FunExpr -> FunExpr
```

First, our elementary functions. You can prove them using the methods described above, but the easiest way to find them is to just look them up in some table of integrals (dust off that old calculus cheat sheet) or on WolframAlpha (or Wikipedia, or whatever. Up to you).

```
integrate Exp = Exp :- Const 1
```

Note that  $\log(x)$  is not defined in  $x = 0$ , so we don't have to add any corrective constant as  $F(0)$  wouldn't make sense anyways.

```
integrate Log = Id :* (Log :- Const 1)
integrate Sin = Const 1 :- Cos
integrate Cos = Sin
integrate Asin = (Const 1 :- Id:^(Const 2)):^(Const 0.5)
                :+ Id :* Asin
                :- Const 0
integrate Acos = Id :* Acos
                :- (Const 1 :- Id:^(Const 2)):^(Const 0.5)
                :+ Const 1
```

These two good boys. Very simple as well.

```
integrate Id = Id:^Const 2 :/ Const 2
integrate (Const d) = Const d :* Id
```

Addition and subtraction is trivial. Just use the backwards method and compare to how sums and differences are differentiated.

```
integrate (f :+ g) = integrate f :+ integrate g
integrate (f :- g) = integrate f :- integrate g
```

Delta is easy. Just expand it to the difference that it is, and integrate.

```
integrate (Delta h f) = integrate (f :. (Id :+ Const h) :- f)
```

A derivative? That's trivial, the integration and differentiation cancel each other, right? Nope, not so simple! To conform to our specification of `integrate` that  $F(0) = 0$ , we have to make sure that the constant coefficient is equal to 0, which it might not be if we just cancel the operations. The simplest way to solve this is to evaluate the function at  $x = 0$ , and check the value. We then subtract a term that corrects the function such that  $I(D(f))[0] = 0$ . We'll write a simple function center for this later.

```
integrate (D f) = center f
```

Integrating an integral? Just integrate the integral!

```
integrate (I f) = integrate (integrate f)
```

Aaaaaand now it starts to get complicated.

There exists a great product rule in the case of differentiation, but not for integration. There just isn't any nice way to integrate a product that always works! The integration rule that's most analogous to the product rule for differentiation, is integration by parts:

$$\int f(x)g(x)dx = f(x)G(x) - \int f'(x)G(x)dx$$

Hmm, this doesn't look quite as helpful as the differentiation product rule, does it? We want this rule to give us an expression of simpler and/or fewer integrals, and it may indeed do so. For example, the integration of the product  $x * e^x$  is a great examples of a case where it works well:

$$\int xe^x dx = xe^x - \int 1e^x dx = xe^x - e^x = e^x(x - 1)$$

Now THAT is a simplification. However, just by flipping the order of the expressions, we get a case where the integration by parts rule only makes things worse:

$$\begin{aligned} \int e^x x dx &= e^x \frac{x^2}{2} - \int e^x \frac{x^2}{2} dx \\ &= e^x \frac{x^2}{2} - (e^x \frac{x^3}{3!} - \int e^x \frac{x^3}{3} dx) \\ &= e^x \frac{x^2}{2} - (e^x \frac{x^3}{3!} - (e^x \frac{x^4}{4!} - \int e^x \frac{x^4}{4!} dx)) \\ &= \dots \end{aligned}$$

Oh no, it's an infinite recursion with successive increase in complexity! Sadly, there's no good way around it. By using heuristics, we could construct a complicated algorithm that guesses the best order of factors in a product when integrating, but that's way out of scope for this book.

Further, as a consequence of Liouville's theorem, the integration by parts rule is simply not defined in the case of  $g(x)$  not being integrable to  $G(x)$ . And so, as there exists no definitely good way to do it in ALL cases, we're forced to settle for a conservative approach.

If we rewrite the formula for integration by parts to use  $g'$  instead of  $g$

$$\int f(x)g'(x)dx = f(x)g(x) - \int f'(x)g(x)dx$$

we see that there are two cases where the integral is well defined:

$$\int f(x)g'(x)dx = f(x)g(x) - \int f'(x)g(x)dx$$

and

$$\int f'(x)g(x)dx = f(x)g(x) - \int f(x)g'(x)dx$$

I.e., if we already know the integral of one factor, we can integrate the product.

```
integrate (D f :* g) = center (f :* g :- integrate (f :* derive g))
integrate (f :* D g) = center (f :* g :- integrate (derive f :* g))
```

Also, we can add a few cases for integrals we know, like multiplication with a constant

```
integrate (Const c :* f) = Const c :* integrate f
```

The rule for quotients is very similar

```
integrate (D f ./ g) =
  center (f ./ g :+: integrate (f :* (D g ./ (g:^Const 2))))
```

There is no good rule for exponentials in general. Only for certain combinations of functions in the base and exponent is symbolic integration well defined. We'll only treat the special case of  $x^c$ , which at least implies that we can use polynomials.

```
integrate (Id :^ Const c) = Id:^(Const (c+1)) ./ Const (c+1)
```

**Exercise.** Find more rules of integrating exponentials and add to the implementation.

▼ **Solution**

/details>

Integration of function composition is, simply said, somewhat complicated. The technique to use is called "integration by substitution", and is something like a reverse of the chain-rule of differentiation. This method is tricky to implement, as the way humans execute this method is highly dependent on intuition and a mind for patterns. A brute-force solution would be possible to implement, but is out of scope for this book, and not really relevant to what we want to learn here. We'll leave symbolic integration of composition undefined.

And if we couldn't integrate the expression as is, first try simplifying it and see if we know how to integrate the new expression. If that fails, just wrap the expression in the `I` constructor, unchanged. This will signify that we don't know how to symbolically integrate the expression. During evaluation, we can use `integrateApprox` to compute the integral numerically.

```
integrate f = let fsim = simplify f
              in if f == fsim
                 then I f
                 else integrate fsim
```

Finally, the helper function `center` to center functions such that  $f(0) = 0$ .

```
center f = f :- Const ((eval f) 0)
```

## The value of evaluation

What comes after construction of function expressions? Well, using them of course!

One way of using a function expression is to evaluate it, and use it just as you would a normal Haskell function. To do this, we need to write an evaluator.

An evaluator simply takes a syntactic representation and returns the semantic value, i.e. `eval :: SYNTAX -> SEMANTICS`.

In the case of our calculus language:

```
eval :: FunExpr -> (RealFunc)
```

To then evaluate a FunExpr is not very complicated. The elementary functions and the Id function are simply substituted for their Haskell counterparts.

```
eval Exp = exp
eval Log = log
eval Sin = sin
eval Cos = cos
eval Asin = asin
eval Acos = acos
eval Id = id
```

Const is evaluated according to the definition  $const(c) = x \mapsto c$

```
eval (Const c) = \x -> c
```

How to evaluate arithmetic operations on functions may not be as obvious, but we just implement them as they were defined earlier in the chapter.

```
eval (f :+: g) = \x -> (eval f x + eval g x)
eval (f :- g) = \x -> (eval f x - eval g x)
eval (f :* g) = \x -> (eval f x * eval g x)
eval (f :/ g) = \x -> (eval f x / eval g x)
eval (f :^ g) = \x -> (eval f x ** eval g x)
```

Function composition is similarly evaluated according to the earlier definition

```
eval (f :. g) = \x -> eval f (eval g x)
```

Delta is just expanded to the difference that it really is

```
eval (Delta h f) = eval (f :. (Id :+: Const h) :- f)
```

For derivatives, we just apply the symbolic operation we wrote, and then evaluate the result.

```
eval (D f) = eval (derive f)
```

With integrals, we first symbolically integrate the expression as far as possible, then apply the numerical integrateApprox if we can't figure out the integral further.

```
eval (I f) = let _F = simplify (integrate f)
             in if _F == (I f)
                then integrateApprox (eval f) 0.001 0
                else eval _F
```

[src: [Calculus/IntegralCalc.lhs](#)] Previous: [Differential calculus](#) [Table of contents](#) Next: [Plotting graphs](#)

© Björn Werner, Erik Sjöström, Johan Johansson, Oskar Lundström (2018), GPL

# Learn You a **Physics** for Great Good!

>>> WORK IN PROGRESS <<<

## Calculus / Plotting graphs

[src: [Calculus/VisVerApp.lhs](#)] Previous: [Integral calculus](#) [Table of contents](#) Next: [Syntax trees](#)

```
module Calculus.VisVerApp where
```

```
import Calculus
```

```
import Hatlab.Plot
```

## Plotting with Hatlab

The brain likes seeing things. Let's give it a good looking reward!

We'll now make combined use of all of our nice functions. `simplify`, `derive`, `integrate`, `eval`, and `show`, all together: the most ambitious crossover event in history!

First, we create some function expressions ready to be shown and evaluated.

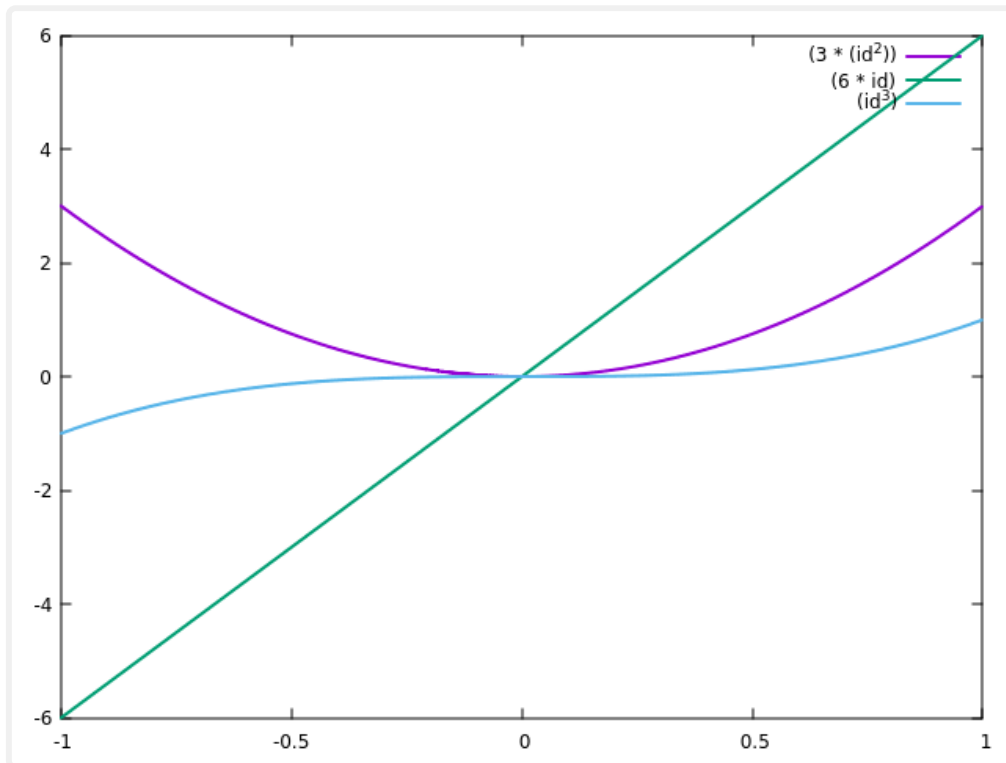
```
f = Const 3 :* Id:^Const 2
f' = simplify (derive f)
_F = simplify (integrate f)
```

Then, we define a helper function to plot a list of function expressions with Hatlab.

```
plotFunExprs :: [FunExpr] -> IO ()
plotFunExprs = plot . fmap (\f -> Fun (eval f) (show f))
```

Now try it for yourself! Let's see the fruits of our labour!

```
ghci> plotFunExprs [f, f', _F]
```



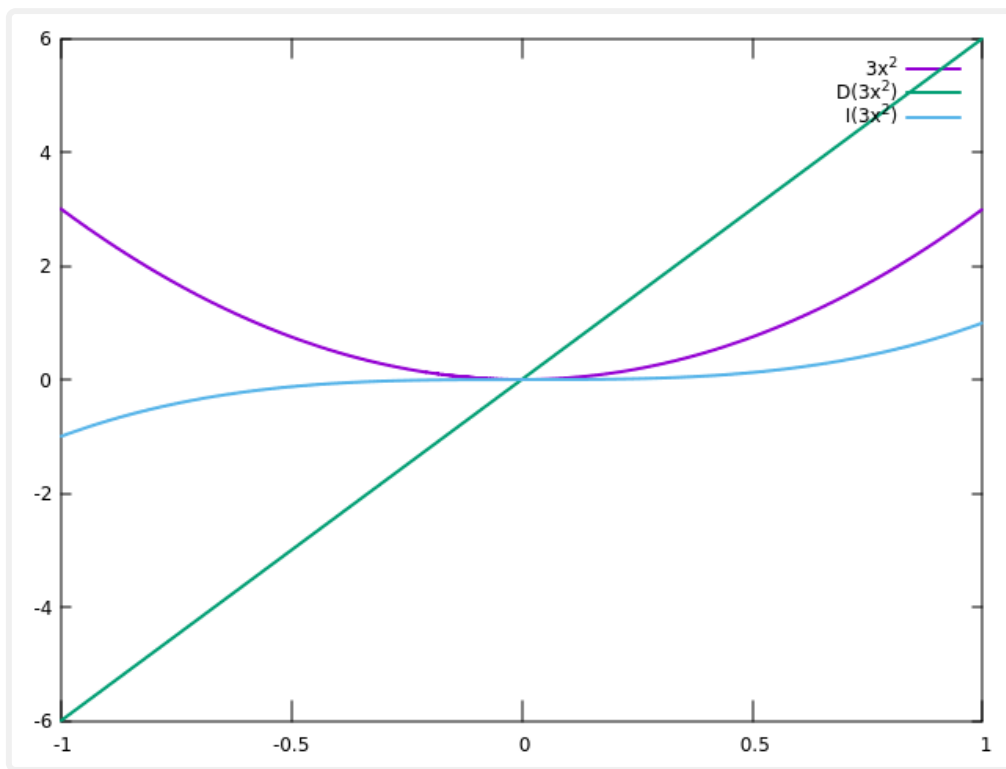
For fun, we can also plot the same functions but using our approximative functions for differentiation and integration

```
g x = 3 * x^2
g' x = deriveApprox g 0.001 x
_G x = integrateApprox g 0.001 0 x
```

Then plot with

```
ghci> plot [Fun g "3x^2", Fun g' "D(3x^2)", Fun _G "I(3x^2)"]
```



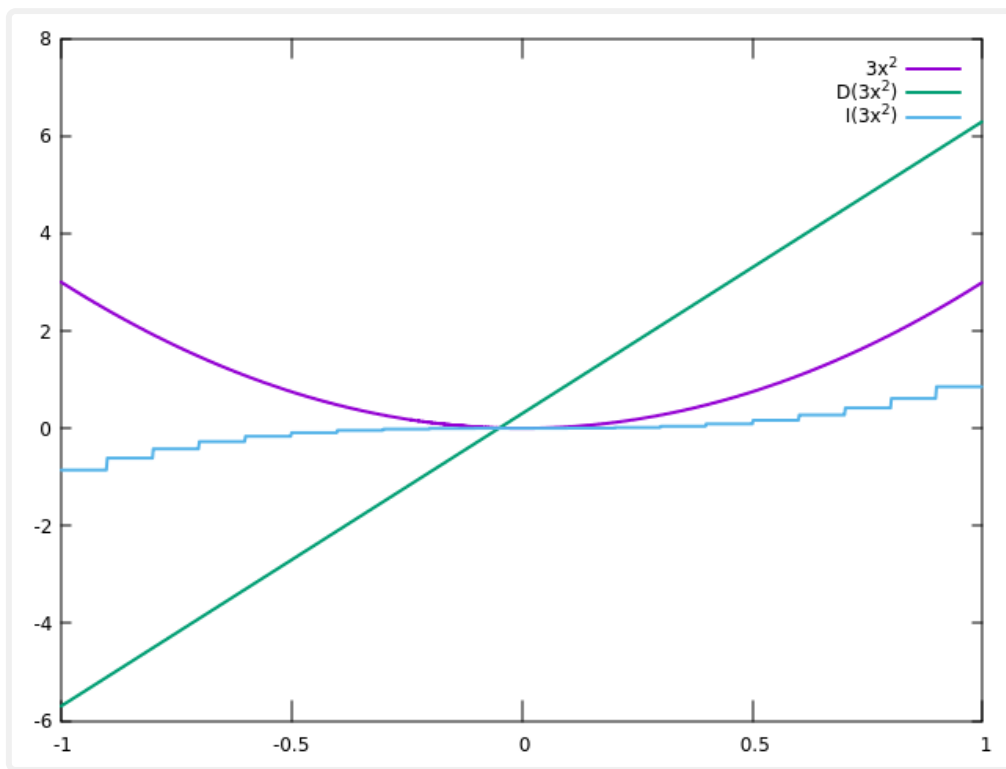


Waddaya know! They look identical! I guess it just goes to show that a good approximation is often good enough.

If we turn down the precision, we start to notice the errors

```
h x = 3 * x^2
h' x = deriveApprox h 0.1 x
_H x = integrateApprox h 0.1 0 x
```

```
ghci> plot [Fun h "3x^2", Fun h' "D(3x^2)", Fun _H "I(3x^2)"]
```



[src: [Calculus/VisVerApp.lhs](#)] Previous: [Integral calculus](#) [Table of contents](#) Next: [Syntax trees](#)

© Björn Werner, Erik Sjöström, Johan Johansson, Oskar Lundström (2018), GPL

# Learn You a **Physics** for Great Good!

>>> WORK IN PROGRESS <<<

## Calculus / Syntax trees

[src: [Calculus/SyntaxTree.lhs](#)] Previous: [Plotting graphs](#) [Table of contents](#) Next: [Vectors](#)

```
module Calculus.SyntaxTree where

import           Calculus.FunExpr
import           Calculus.DifferentialCalc
import           Calculus.IntegralCalc
```

A fun and useful way to visualize expressions is to model them as trees. In our case we want to model *FunExpr* where the nodes and leaves will be our constructors.

In order to do this will import two packages, one for constructing trees and one for pretty printing them.

```
import           Data.Tree           as T
import           Data.Tree.Pretty    as P
```

Now we can construct the function that takes a *FunExpr* and builds a tree from it. Every node is a string representation of the constructor and a list of its sub trees (branches).

```
makeTree :: FunExpr -> Tree String
makeTree (e1 :+: e2) = Node "+" [makeTree e1, makeTree e2]
makeTree (e1 :- e2) = Node "-" [makeTree e1, makeTree e2]
makeTree (e1 :* e2) = Node "*" [makeTree e1, makeTree e2]
makeTree (e1 :/ e2) = Node "Div" [makeTree e1, makeTree e2]
makeTree (e1 ^ e2) = Node "**" [makeTree e1, makeTree e2]
makeTree (e1 :. e2) = Node "o" [makeTree e1, makeTree e2]
makeTree (D e)      = Node "d/dx" [makeTree e]
makeTree (Delta r e) = Node "Δ" [makeTree (Const r), makeTree e]
```

```

makeTree (I e)      = Node "I"    [makeTree e]
makeTree Id        = Node "Id"   []
makeTree Exp       = Node "Exp"  []
makeTree Log       = Node "Log"  []
makeTree Sin       = Node "Sin"  []
makeTree Cos       = Node "Cos"  []
makeTree Asin      = Node "Asin" []
makeTree Acos      = Node "Acos" []
makeTree (Const num) = Node (show num) [] --(show (floor num)) [] -- | Note the use of
  floor

```

Now we construct trees from our expressions but we still need to print them out. For this we'll use the function `drawVerticalTree` which does exactly what its name suggests. We can then construct a function to draw expressions.

```

printExpr :: FunExpr -> IO ()
printExpr = putStrLn . drawVerticalTree . makeTree

```

Now let's construct a mildly complicated expression

```

e = Delta 3 (Delta (negate 5) (I Acos) :. (Acos :* Exp))

```

And print it out.

```

ghci > printExpr e
      Δ
      |
  -----
 /          \
3            0
            |
  -----
 /          \
Δ            *
|            |
-----
 /  \      /  \
-5  I  Acos  Exp
    |
    Acos

```

Pretty prints the steps taken when canonifying an expression

```

prettyCan :: FunExpr -> IO ()
prettyCan e =
  let t = makeTree e
      e' = canonify e
      t' = makeTree e'
  in if t == t' then putStrLn $ drawVerticalTree t
      else do
        putStrLn $ drawVerticalTree t
        prettyCan e'

```

Pretty prints syntactic checking of equality

```

prettyEqual :: FunExpr -> FunExpr -> IO Bool
prettyEqual e1 e2 = if e1 == e2 then
  do
    putStrLn "It's equal!"
    putStrLn $ drawVerticalForest [makeTree e1, makeTree e2]
    return True
  else do
    putStrLn "Not equal -> Simplifying"
    putStrLn $ drawVerticalForest [makeTree e1, makeTree e2]
    let c1 = canonify e1
        c2 = canonify e2
    in if c1 == e1 && c2 == e2 then putStrLn "Can't simplify no more"
        >> return False
        else prettyEqual c1 c2

```

Syntactic checking of equality

```

equal :: FunExpr -> FunExpr -> Bool
equal e1 e2 = (e1 == e2) ||
  (let c1 = canonify e1
      c2 = canonify e2
   in (not (e1 == c1 && c2 == e2) && equal c1 c2))

```

Parse an expression as a Tree of Strings

Of course this is all bit too verbose, but I'm keeping it that way until every case is covered, Calculus is a bit of a black box for me right now

```

canonify :: FunExpr -> FunExpr

```

Addition

```
canonify (e :+: Const 0)      = canonify e
canonify (Const 0 :+: e)      = canonify e
canonify (Const x :+: Const y) = Const (x + y)
canonify (e1 :+: e2)          = canonify e1 :+: canonify e2
```

## Subtraction

```
canonify (e :- Const 0)      = canonify e
canonify (Const a :- Const b) = Const (a - b)
canonify (e1 :- e2)          = canonify e1 :- canonify e2
```

## Multiplication

```
canonify (_ :* Const 0)      = Const 0
canonify (Const 0 :* _)      = Const 0
canonify (e :* Const 1)      = canonify e
canonify (Const 1 :* e)      = canonify e
canonify (Const a :* Const b) = Const (a * b)
canonify (e1 :* e2)          = canonify e1 :* canonify e2
```

## Division

```
canonify (Const a :/ Const b) = Const (a / b)
canonify (e1 :/ e2)            = canonify e1 :/ canonify e2
```

## Delta

```
canonify (Delta r e)          = Delta r $ canonify e
```

## Derivatives

```
canonify (D e)                = derive e
```

## Composition

```
canonify (e1 :: e2)           = canonify e1 :: canonify e2
```

## Catch all

```
canonify e                     = e
```

# “Proofs”

```
syntacticProofOfComForMultiplication :: FunExpr -> FunExpr -> IO Bool
syntacticProofOfComForMultiplication e1 e2 = prettyEqual (e1 :* e2) (e2 :* e1)
```

```
syntacticProofOfAssocForMultiplication :: FunExpr -> FunExpr -> FunExpr -> IO Bool
syntacticProofOfAssocForMultiplication e1 e2 e3 = prettyEqual (e1 :* (e2 :* e3))
                                                         ((e1 :* e2) :* e3)
```

```
syntacticProofOfDistForMultiplication :: FunExpr -> FunExpr -> FunExpr -> IO Bool
syntacticProofOfDistForMultiplication e1 e2 e3 = prettyEqual (e1 :* (e2 :+: e3))
                                                         ((e1 :* e2) :+: (e1 :*
e3))
```

```
{- syntacticProofOfIdentityForMultiplication :: FunExpr -> IO Bool -}
{- syntacticProofOfIdentityForMultiplication e = -}
{- putStrLn "[*] Checking right identity" >> -}
{- prettyEqual e (1 :* e) >> -}
{- putStrLn "[*] Checking left identity" >> -}
{- prettyEqual e (e :* 1) -}
```

```
{- syntacticProofOfPropertyOf0ForMultiplication :: FunExpr -> IO Bool -}
{- syntacticProofOfPropertyOf0ForMultiplication e = -}
{- prettyEqual (e :* 0) 0 -}
```

```
-- | Fails since default implementation of negate x for Num is 0 - x
{- syntacticProofOfPropertyOfNegationForMultiplication :: FunExpr -> IO Bool -}
{- syntacticProofOfPropertyOfNegationForMultiplication e = -}
{- prettyEqual (Const (-1) :* e) (negate e) -}
```

```
syntacticProofOfComForAddition :: FunExpr -> FunExpr -> IO Bool
syntacticProofOfComForAddition e1 e2 = prettyEqual (e1 :+: e2) (e2 :+: e1)
```

```
syntacticProofOfAssocForAddition :: FunExpr -> FunExpr -> FunExpr -> IO Bool
syntacticProofOfAssocForAddition e1 e2 e3 = prettyEqual (e1 :+: (e2 :+: e3))
                                                         ((e1 :+: e2) :+: e3)
```

```
test :: FunExpr -> FunExpr -> IO Bool
test b c = prettyEqual b (a :* c)
```

**where**

`a = b :/ c`

```
syntacticProofOfIdentityForAddition :: FunExpr -> IO Bool
```

```
syntacticProofOfIdentityForAddition e = putStrLn "[*] Checking right identity" >>  
  prettyEqual e (0 :+: e) >>  
    putStrLn "[*] Checking left identity" >>  
      prettyEqual e (e :+: 0)
```

## Dummy expressions

```
e1 = Const 1
```

```
e2 = Const 2
```

```
e3 = Const 3
```

```
e4 = (Const 1 :+: Const 2) :* (Const 3 :+: Const 4)
```

```
e5 = (Const 1 :+: Const 2) :* (Const 4 :+: Const 3)
```

```
e6 = Const 2 :+: Const 3 :* Const 8 :* Const 19
```

[src: [Calculus/SyntaxTree.lhs](#)]    Previous: [Plotting graphs](#)    [Table of contents](#)    Next: [Vectors](#)

© Björn Werner, Erik Sjöström, Johan Johansson, Oskar Lundström (2018), GPL



# Learn You a **Physics** for Great Good!

>>> WORK IN PROGRESS <<<

## Linear algebra / Vectors

[src: [Vector/Vector.lhs](#)]

Previous: [Syntax trees](#)

[Table of contents](#)

Next: [Introduction](#)

```
module Vector.Vector where
import Test.QuickCheck hiding (scale)
```

### Vectors in two dimensions.

A physical quantity that has only a magnitude is called a scalar. In Haskell we'll represent this using a Double.

```
type Scalar = Double
```

A vector is a quantity that has both a magnitude and a direction. For instance the *velocity* of a moving body involves its speed (magnitude) and the direction of motion.

We can represent the direction of a vector in two dimensions using its  $x$  and  $y$  coordinates, which are both scalars. The direction is then given by the angle between these coordinates and the origin (0,0).

In order to maintain generalizability and useability in the next chapters we'll use a typesynonym in the data declaration but throughout this whole chapter we'll use Scalars as we've defined them here.

```
data Vector2 num = V2 num num
```

We can even introduce a type that makes that specific relationship clearer:

```
type VectorTwo = Vector2 Scalar
```

The magnitude of the vector is it's length. We can calculate this using Pythagorean theorem:

$$x^2 + y^2 = \text{magnitude}^2$$

In haskell this would be:

```
magnitude :: VectorTwo -> Scalar
magnitude (V2 x y) = sqrt (x^2 + y^2)
```

And now we can calculate the magnitude of a vector in two dimensions:

```
Vector> let vec = V2 5 3
Vector> magnitude vec
5.830951894845301
```

Addition and subtraction of vectors is accomplished using the components of the vectors. For instance when adding the forces (vectors) acting on a body we would add the components of the forces acting in the  $x$  direction and the components in the  $y$  direction. So our functions for adding and subtracting vectors in two dimensions are:

```
add :: VectorTwo -> VectorTwo -> VectorTwo
add (V2 x1 y1) (V2 x2 y2) = V2 (x1 + x2) (y1 + y2)

sub :: VectorTwo -> VectorTwo -> VectorTwo
sub (V2 x1 y1) (V2 x2 y2) = V2 (x1 - x2) (y1 - y2)
```

But this only works for two vectors. In reality we might be working with several hundreds of vectors so it would be useful to add, for instance a list of vectors together and get one final vector as a result. We can use *foldr* using the zero vector as a starting value. to accomplish this.

```
zeroVector :: VectorTwo
zeroVector = V2 0 0

addListOfVectors :: [VectorTwo] -> VectorTwo
addListOfVectors = foldr add zeroVector
```

Let's try it out!

```
*Vector> let vec1 = V2 5 3
*Vector> let vec2 = V2 6 5
*Vector> sub vec1 vec2
```

```
<interactive>:23:1: error:
```

- No instance for (Show VectorTwo) arising from a use of 'print'
- In a stmt of an interactive GHCi command: print it

The interpreter is complaining that it doesn't know how to interpret our datatype for vectors as a string. The easy solution would be to just derive our instance for Show, but to really solidify the fact that we are working with coordinates let's make our own instance for Show.

```
instance Show num => Show (Vector2 num) where
  show (V2 x y) = "(" ++ show x ++ ", " ++ show y ++ ")"
```

And let's try our example again:

```
*Vector> let vec1 = V2 5 3
*Vector> let vec2 = V2 6 5
*Vector> sub vec1 vec2
(-1 x, -2 y)
```

And let's also try adding a list of vectors using our new function:

```
*Vector> let vec3 = V2 8 9
*Vector> let vectors = [vec1, vec2, vec3]
*Vector> addListOfVectors vectors
(19.0 x, 17.0 y)
```

It works!

We can also multiply a vector by a scalar. This is also done componentwise. We'll call this scaling a vector. So we could double a vector by multiplying it with 2.0 and halving it by multiplying it with 0.5.

```
scale :: Scalar -> VectorTwo -> VectorTwo
scale factor (V2 x y) = V2 (factor * x) (factor * y)
```

Combining this with the unit vectors:

```
unitX :: VectorTwo
unitX = V2 1 0

unitY :: VectorTwo
unitY = V2 0 1
```

We get a new way of making vectors, namely by scaling the unit vectors and adding them together. Let's create the vector (5 x, 3 y) using this approach.

```
*Vector> add (scale 5 unitX) (scale 3 unitY)
(5.0 x, 3.0 y)
```

In order to check that this vector is actually equal to the vector created using the constructor `V2` we need to make our vector an instance of `Eq`.

```
instance Eq num => Eq (Vector2 num) where
  (V2 x1 y1) == (V2 x2 y2) = (x1 == x2) && (y1 == y2)
```

Let's try it out:

```
*Vector> let vec1 = V2 5 3
*Vector> let vec2 = add (scale 5 unitX) (scale 3 unitY)
*Vector> vec1 == vec2
True
```

We have one final important operation left to define for vectors in two dimensions, the dot product. The formula is quite simple:

$$\vec{a} \cdot \vec{b} = a_x \cdot b_x + a_y \cdot b_y$$

And our function simply becomes:

```
dotProd :: VectorTwo -> VectorTwo -> Scalar
dotProd (V2 ax ay) (V2 bx by) = ax * bx + ay * by
```

But this doesn't give us any intuition about what it means to take the dot product between vectors. The common interpretation is "geometric projection", but that only makes sense if you already understand the dot product. Let's try to give an easier analogy using the dash panels (boost pads) from *Mario Kart*. The dash panel is designed to give you boost of speed in a specific direction, usually straight forward. So the vector associated with the

dash panel can be represented with a unit vector multiplied with some factor of boost, say 10.

```
dashPanel :: VectorTwo
dashPanel = scale 10 unitY
```

Now let's say that your cart has this arbitrarily chosen velocity vector:

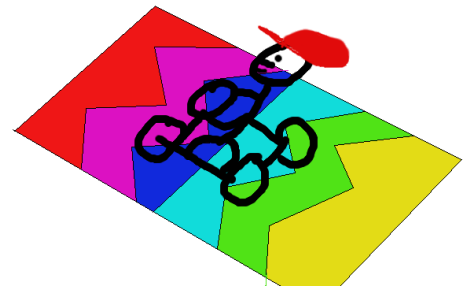
```
cart :: VectorTwo
cart = V2 3 5
```

Depending on which angle you hit the dash panel you'll receive different amounts of boost. Since the  $x$ -component of the dashPanel is 0 any component of speed on the  $x$ -direction will be reduced to zero. Only the speed in the direction of  $y$  will be boosted. But there are worse ways to hit the dash panel. We could for instance create a new velocity vector with the exact same magnitude of speed but which would receive a worse boost.

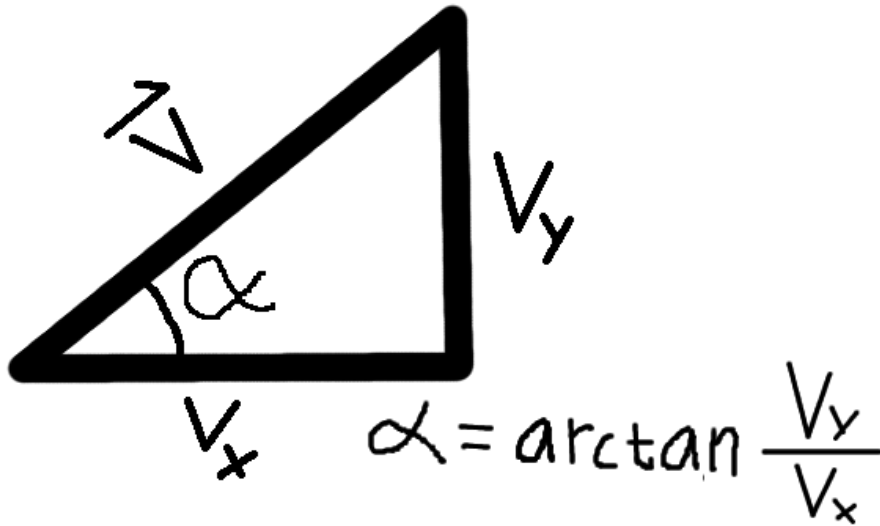
```
worseCart :: VectorTwo
worseCart = V2 5 3
```

Let's see this in action.

```
*Vector> magnitude cart == magnitude worseCart
True
*Vector> dotProd dashPanel cart
50.0
*Vector> dotProd dashPanel worseCart
30.0
```



We talked a lot about angles between vectors but we haven't used it in our code, so let's make a function which calculates the angle of a vector. The formula is as follows:



We'll use Doubles to represent the angle.

```
type Angle = Double
```

```
angle :: VectorTwo -> Scalar
```

```
angle (V2 x y) = atan y/x
```

Using angles and magnitudes we can even write a new function for making vectors:

```
mkVector :: Scalar -> Angle -> VectorTwo
```

```
mkVector mag angle = V2 x y
```

```
where
```

```
  x = mag * cos angle
```

```
  y = mag * sin angle
```

## Vectors in three dimensions.

The datatype for a vector in three dimensions is basically the same as vector in two dimensions, we'll just add a *z*-component.

Again we will generalize this over a type synonym

```
data Vector3 num = V3 num num num
```

And a type synonym that we can use throughout the rest of this chapter.

```
type VectorThree = Vector3 Scalar
```

Similarly the functions for adding three dimensional vectors:

```
add :: VectorThree -> VectorThree -> VectorThree
add (V3 x1 y1 z1) (V3 x2 y2 z2) = V3 (x1 + x2) (y1 + y2) (z1 + z2)
```

Multiplying with a scalar:

```
scale3 :: Scalar -> VectorThree -> VectorThree
scale3 fac (V3 x y z) = V3 (fac * x) (fac * y) (fac * z)
```

And for calculating the magnitude:

```
mag3 :: VectorThree -> Scalar
mag3 (V3 x y z) = sqrt (x**2 + y**2 + z**2)
```

Looks early similar to our functions for vectors in two dimensions. This suggest that there might be a better way to handle this, in order to avoid repeating ourselves.

Addition and subtraction on vectors works by “unpacking” the vectors, taking their components, applying some function to them (+/-) and then packing them up as a new vector. This is very similar to the Haskell function *zipWith* which works over lists instead of vectors.

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

When we’re multiplying with a scalar we again unpack the vector and then apply multiplication with a factor to each component before packing it up again. This is quite similar to the Haskell function *map*, which again works over lists.

```
map :: (a -> b) -> [a] -> [b]
```

When calculating the magnitude of a vector we first unpack the vector and then apply <sup>2</sup> to each component of the vector. This is doable with aforementioned *map*. We then *fold* the components together using  $+$  which results in a final scalar value. Those of you familiar with functional languages will know where I’m going with this, those of you who aren’t will hopefully understand where I’m going when reading the examples.

Using this information we can now create a new class for vectors which implement this functionality:

```
class Vector vector where
  vmap      :: (num -> num)          -> vector num -> vector num
  vzipWith  :: (num -> num -> num) -> vector num -> vector num -> vector num
  vfold     :: (num -> num -> num) -> vector num -> num
```

Now we have a blueprint for what vector is, so let's implement it for our own vector datatypes.

```
instance Vector Vector2 where
  vmap      f (V2 x y)          = V2 (f x) (f y)
  vzipWith  f (V2 x y) (V2 x' y') = V2 (f x x') (f y y')
  vfold     f (V2 x y)          = f x y
```

```
instance Vector Vector3 where
  vmap      f (V3 x y z)          = V3 (f x) (f y) (f z)
  vzipWith  f (V3 x y z) (V3 x' y' z') = V3 (f x x') (f y y') (f z z')
  vfold     f (V3 x y z)          = f z $ f x y
```

Now we're finally leveraging the power of the Haskell typesystem!

We can now implement more generalized functions for addition and subtraction between vectors.

```
add :: (Num num, Vector vec) => vec num -> vec num -> vec num
add = vzipWith (+)
```

```
sub :: (Num num, Vector vec) => vec num -> vec num -> vec num
sub = vzipWith (-)
```

For multiplying with a scalar:

```
scale :: (Num num, Vector vec) => num -> vec num -> vec num
scale factor = vmap (* factor)
```

And for calculating the magnitude of a vector:

```
magnitude :: (Floating num, Vector vec) => vec num -> num
magnitude = sqrt . vfold (+) . vmap (**2)
```



We can even use it to make a generalized function for calculating the dot product.

```
dotProd :: (Num num, Vector vec) => vec num -> vec num -> num
dotProd v1 v2 = vfold (+) $ vzipWith (*) v1 v2
```

## Cross Product

We have one final function left to define, the cross product. The formula is as follows:

$$\vec{a} \times \vec{b} = |\vec{a}| \cdot |\vec{b}| \cdot \sin(\theta)$$

Where  $\theta$  is the angle between the vectors. And  $|\vec{a}|$ ,  $|\vec{b}|$  are the magnitudes of the vectors.

So our function for calculating the cross product becomes:

TODO: Generate normal vector as well. Codify right hand rule

```
crossProd :: Vector3 -> Vector3 -> Vector3
crossProd a b = (magnitude a) * (magnitude b) * sin (angleBetween a b)
  where
    angleBetween :: (Vector vec) => vec -> vec -> Scalar
    angleBetween v1 v2 = acos ((dotProd v1 v2) / ((magnitude v1) * (magnitude v2)))
```

Working cross product using matrix rules.

```
crossProd :: Num num => Vector3 num -> Vector3 num -> Vector3 num
crossProd (V3 x y z) (V3 x' y' z') = V3 (y*z' - z*y') -- X
                                         (z*x' - x*z') -- Y
                                         (x*y' - y*x') -- Z
```

## Quickcheck!

There are certain laws or properties that vectors adhere to, for example the jacobi identity:

$$\vec{a} \times (\vec{b} \times \vec{c}) + \vec{b} \times (\vec{c} \times \vec{a}) + \vec{c} \times (\vec{a} \times \vec{b}) = 0$$

Or that the cross product is anticommutative. We can't actually prove these in a meaningful way without a whole bunch of packages and pragmas, but we can quickcheck

them. But to do that we need to be able to generate vectors, so let's make our vectors an instance of *Arbitrary*.

We do this by generating arbitrary scalars and then constructing vectors with them.

```
instance Arbitrary num => Arbitrary (Vector2 num) where
  arbitrary = arbitrary >>= (\(s1, s2) -> return $ V2 s1 s2)

instance Arbitrary num => Arbitrary (Vector3 num) where
  arbitrary = arbitrary >>= (\(s1, s2, s3) -> return $ V3 s1 s2 s3)
```

Let's try it out!

```
ghci> generate arbitrary :: IO (Vector2 Scalar)
(-26.349975377051404 x, 9.71134047527185 y)
```

Seems pretty random to me.

Now we can check some properties, let's start with commutativity of vector addition:

$$\vec{a} + \vec{b} = \vec{b} + \vec{a}$$

Which translates to:

```
prop_CommutativityAddition :: VectorThree -> VectorThree -> Bool
prop_CommutativityAddition v1 v2 = v1 + v2 == v2 + v1
```

And we test this in the *repl*.

```
ghci> quickCheck prop_CommutativityAddition
+++ OK, passed 100 tests.
```

And associativity of addition:

$$\vec{a} + (\vec{b} + \vec{c}) = (\vec{a} + \vec{b}) + \vec{c}$$

```
prop_AssociativityAddition :: VectorThree -> VectorThree -> VectorThree -> Bool
prop_AssociativityAddition a b c = a + (b + c) == (a + b) + c
```

```
ghci> quickCheck prop_AssociativityAddition
*** Failed! Falsifiable (after 2 tests):
(0.5240133611343812 x, -0.836882545823441 y, -4776.775557184785 z)
```

```
(-0.17261751005585407 x, 0.7893754200476363 y, -0.19757165887775568 z)
(0.3492200657348603 x, 0.10861834028920295 y, 0.45838513657221946 z)
```

This is very strange since the laws should always be correct. But this error stems from the fact that we're using a computer and that using doubles (Scalar) will introduce approximation errors. We can fix this by relaxing our instance for *Eq* and only requiring the components of the vectors to be approximately equal.

% TODO: Equation

```
eps :: Floating num => num
eps = 1 * (10 ** (-5))

instance (Floating num, Eq num, Ord num) => Eq (Vector2 num) where
  (V2 x1 y1) == (V2 x2 y2) = xCheck && yCheck
  where
    xCheck = abs (x1 - x2) <= eps
    yCheck = abs (y1 - y2) <= eps
```

Let's try again.

```
*Vector.Vector> quickCheck prop_AssociativityAddition
+++ OK, passed 100 tests.
```

## More laws

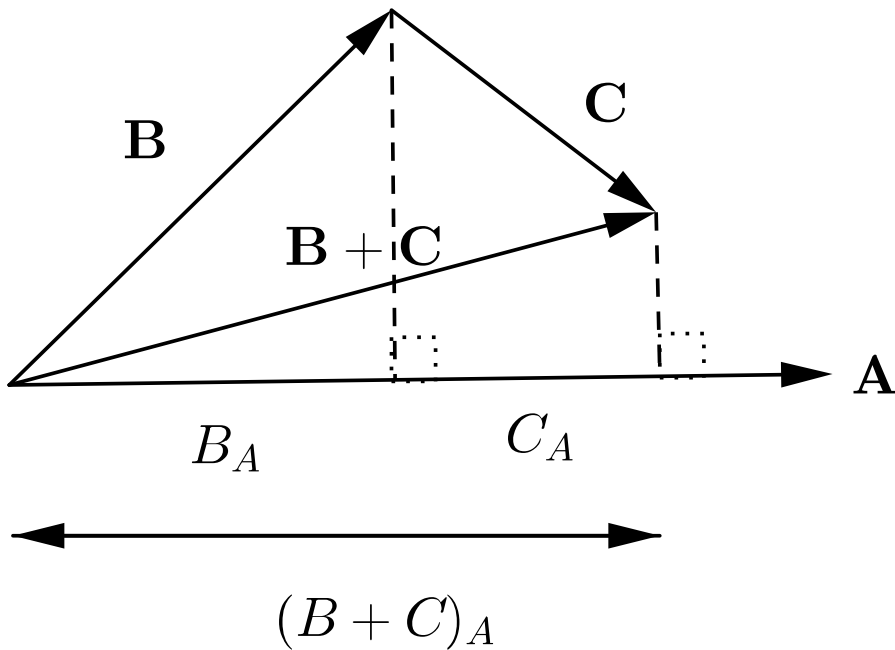
Dot product is commutative:

```
prop_dotProdCommutative :: VectorThree -> VectorThree -> Bool
prop_dotProdCommutative a b = dotProd a b == dotProd b a
```

In order to check some laws which depends on checking the equality of scalars we'll introduce a function which checks that two scalars are approximately equal.

```
-- Approx equal
(~=) :: Scalar -> Scalar -> Bool
rhs ~= lhs = abs (rhs - lhs) <= eps
```

Dot product is distributive over addition:



$$\vec{a} \cdot (\vec{b} + \vec{c}) = (\vec{a} \cdot \vec{b}) + (\vec{a} \cdot \vec{c})$$

```
prop_dotProdDistributiveAddition :: VectorThree -> VectorThree -> VectorThree -> Bool
prop_dotProdDistributiveAddition a b c = dotProd a (b + c) == (dotProd a b + dotProd a c)
```

The dot product is homogeneous under scaling in each variable:

$$(x * \vec{a}) \cdot b = x * (\vec{a} \cdot b) = \vec{a} \cdot (x * \vec{b})$$

```
prop_dotProdHomogeneousScaling :: Scalar -> VectorThree -> VectorThree -> Bool
prop_dotProdHomogeneousScaling x a b = e1 == e2 && e2 == e3
  where
    e1 = dotProd (scale x a) b
    e2 = x * dotProd a b
    e3 = dotProd a (scale x b)
```

The cross product of a vector with itself is the zero vector.

```
prop_crossProd_with_self :: VectorThree -> Bool
prop_crossProd_with_self v = crossProd v v == 0
```

The crossproduct is anticommutative:

$$\vec{a} \times \vec{b} = -(\vec{b} \times \vec{a})$$

```
prop_crossProdAntiCommutative :: VectorThree -> VectorThree -> Bool
prop_crossProdAntiCommutative v1 v2 = v1 * v2 == - (v2 * v1)
```

The cross product is distributive over addition:

$$\vec{a} \times (\vec{b} + \vec{c}) = (\vec{a} \times \vec{b}) + (\vec{a} \times \vec{c})$$

```
prop_crossProdDistributiveAddition :: VectorThree -> VectorThree -> VectorThree ->
  Bool
prop_crossProdDistributiveAddition a b c = a * (b + c) == (a * b) + (a * c)
```

Vector triple product (Lagrange's formula).

$$\vec{a} \times (\vec{b} \times \vec{c}) = \vec{b}(\vec{a} \cdot \vec{c}) - \vec{c}(\vec{a} \cdot \vec{b})$$

```
prop_lagrange :: VectorThree -> VectorThree -> VectorThree -> Bool
prop_lagrange a b c = a * (b * c) == (scale (dotProd a c) b -
  scale (dotProd a b) c)
```

The Jacobi identity:

$$\vec{a} \times (\vec{b} \times \vec{c}) + \vec{b} \times (\vec{c} \times \vec{a}) + \vec{c} \times (\vec{a} \times \vec{b})$$

```
prop_JacobiIdentity :: VectorThree -> VectorThree -> VectorThree -> Bool
prop_JacobiIdentity a b c = a * (b * c) +
  b * (c * a) +
  c * (a * b) == 0
```

## Fun instances

```
instance Num num => Monoid (Vector2 num) where
  mempty = zeroVector
  mappend = (+)
  mconcat = foldr mappend mempty
```

```
instance Num num => Monoid (Vector3 num) where
  mempty = zeroVector
  mappend = (+)
  mconcat = foldr mappend mempty
```

```

instance Num num => Num (Vector2 num) where
  (+)          = vzipWith (+)
  (*)          = undefined -- Crossproduct not defined for Vector2
  abs          = vmap abs
  negate       = vmap (*(-1))
  -- | Signum can be thought of as the direction of a vector
  signum       = vmap signum
  fromInteger i = V2 (fromInteger i) 0

```

```

instance Num num => Num (Vector3 num) where
  (+)          = vzipWith (+)
  (*)          = crossProd
  abs          = vmap abs
  negate       = vmap (*(-1))
  -- | Signum can be thought of as the direction of a vector
  signum       = vmap signum
  fromInteger i = V3 (fromInteger i) 0 0

```

```
-- TODO: Explain why this works
```

```

zeroVector :: (Vector vec, Num (vec num)) => vec num
zeroVector = 0

```

```

instance Show num => Show (Vector3 num) where
  show (V3 x y z) = "(" ++ show x ++ " x, "
                  ++ show y ++ " y, "
                  ++ show z ++ " z)"

```

```

instance (Floating num, Ord num) => Ord (Vector2 num) where
  compare v1 v2 = compare (magnitude v1) (magnitude v2)

```

```

instance (Floating num, Ord num) => Ord (Vector3 num) where
  compare v1 v2 = compare (magnitude v1) (magnitude v2)

```

```

instance (Ord num, Floating num, Eq num) => Eq (Vector3 num) where
  (V3 x y z) == (V3 x' y' z') = xCheck && yCheck && zCheck
  where
    xCheck = abs (x - x') <= eps
    yCheck = abs (y - y') <= eps
    zCheck = abs (z - z') <= eps

```

```

runTests :: IO ()
runTests = do
  putStrLn "Commutativity of vector addition:"
  quickCheck prop_CommutativityAddition

```

```
putStrLn "Associativity of vector addition:"
quickCheck prop_AssociativityAddition
putStrLn "Dot product distributive over addition:"
quickCheck prop_dotProdDistributiveAddition
putStrLn "Homogeneous scaling:"
quickCheck prop_dotProdHomogeneousScaling
putStrLn "Commutative dot product:"
quickCheck prop_dotProdCommutative
putStrLn "Crossproduct of a vector with itself:"
quickCheck prop_crossProd_with_self
putStrLn "Cross product is anticommutative"
quickCheck prop_crossProdAntiCommutative
putStrLn "Cross product distributive over addition"
quickCheck prop_crossProdDistributiveAddition
putStrLn "Lagrange formula"
quickCheck prop_lagrange
putStrLn "Jacobi identity"
quickCheck prop_JacobiIdentity
```

[src: [Vector/Vector.lhs](#)]    Previous: [Syntax trees](#)    [Table of contents](#)    Next: [Introduction](#)

© Björn Werner, Erik Sjöström, Johan Johansson, Oskar Lundström (2018), GPL

# Learn You a **Physics** for Great Good!

>>> WORK IN PROGRESS <<<

## Dimensions / Introduction

[src: [Dimensions/Intro.lhs](#)] Previous: [Vectors](#) [Table of contents](#) Next: [Value-level dimensions](#)

## Introduction

`module Dimensions.Intro where`

This chapter is about dimensions, quantities and units. What's the difference?

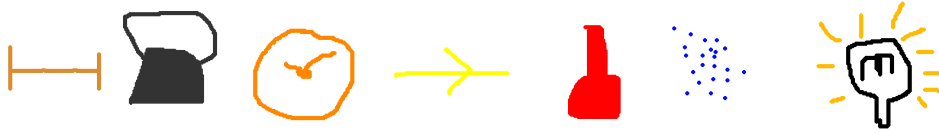
- A **dimension** is “what type of thing something is”. For instance, *length* is a thing, *area* is a thing and *velocity* is a thing. Furthermore, they are *different* things.
- A **quantity** is something that can be *quantified*, i.e, something that can have a number associated with it. Examples are the distance between Stockholm and Gothenburg, the area of a soccer field and the speed of light.
- A **unit** is a certain magnitude of something. 1 metre is for instance approximately the length of your arm.

What's the relation between these? A *quantity* has a *dimension*. The number describing the distance between Stockholm and Gothenburg is of the type length. A *quantity* also has a *unit* that relates the number to a known definite distance. The unit of a quantity must describe the dimension of the quantity. It's not possible to describe a distance with joule. However, describing a distance is possible with both metres and inches. Those are two different units describing a quantity of the same dimension.

The dimension of a quantity is often implicitly understood given its unit. If I have a rope of 1 metre, you know it's a length I'm talking about.



There are 7 *base dimensions*, each with a corresponding SI-unit.



- Length (metre)
- Mass (kilogram)
- Time (seconds)
- Electric current (ampere)
- Temperature (kelvin)
- Amount of substance (mole)
- Luminous intensity (candela)

The outline of this chapter is to first introduce dimensions on *value-level* (to print them nicely). Then we'll do dimensions on *type-level* (to only permit legal operations). And finally we'll combine those results to create a data type for quantities.

In science, SI-units are preferred over all other units. Therefore we'll only care about SI-units. Given this decision, we now have a one-to-one correspondence between dimensions and units, which means that only one concept is really needed!

Let's start with value-level dimensions.

[src: [Dimensions/Intro.lhs](#)] Previous: [Vectors](#) [Table of contents](#) Next: [Value-level dimensions](#)

© Björn Werner, Erik Sjöström, Johan Johansson, Oskar Lundström (2018), GPL

# Learn You a **Physics** for Great Good!

>>> WORK IN PROGRESS <<<

## Dimensions / Value-level dimensions

[src: [Dimensions/ValueLevel.lhs](#)] Previous: [Table of contents](#) Next: [Testing of value-level dimensions](#)

## Value-level dimensions

```
module Dimensions.ValueLevel
  ( Dim(..)
  , mul
  , div
  , length
  , mass
  , time
  , current
  , temperature
  , substance
  , luminosity
  , one
  ) where

import Prelude hiding (length, div)
```

From the introduction, two things became apparanent:

1. Given the unit of a quantity, its dimension is known implicitly.
2. If we only work with SI-units, there is a one-to-one correspondence between dimensions and units.

We'll use these facts when implementing dimensions. More precisely, "length" and "metre" will be interchangeable, and so on.

A dimension can be seen as a product of the base dimensions, with an individual exponent on each base dimension. Because the 7 base dimensions are known in advance, we can design our data type using this fact.

```
data Dim = Dim Integer -- Length
          Integer -- Mass
          Integer -- Time
          Integer -- Current
          Integer -- Temperature
          Integer -- Substance
          Integer -- Luminosity

deriving (Eq)
```

Each field denotes the exponent for the corresponding base dimension. If the exponent is 0, the base dimension is not part of the dimension. Some examples should clarify.

```
length      = Dim 1 0 0 0 0 0 0
mass        = Dim 0 1 0 0 0 0 0
time        = Dim 0 0 1 0 0 0 0
current     = Dim 0 0 0 1 0 0 0
temperature = Dim 0 0 0 0 1 0 0
substance   = Dim 0 0 0 0 0 1 0
luminosity  = Dim 0 0 0 0 0 0 1

velocity    = Dim 1 0 (-1) 0 0 0 0
```

Velocity is  $m/s$  or equivalently  $m^1 * s^{-1}$ . This explains why the exponents are as above.

Noticed how we used "m" (for metre) for implicitly referring to the dimension "length"? It's quite natural to work this way.

**Exercise** Create values for acceleration, area and charge.

### ▼ Solution

```
acceleration = Dim 1 0 (-2) 0 0 0 0
area          = Dim 2 0 0    0 0 0 0
charge        = Dim 0 0 1    1 0 0 0
```

# Multiplication and division

Dimensions can be multiplied and divided. Velocity is, as we just saw, a division between length and time. Multiplication and division of dimensions are performed as if they were regular numbers, or variables holding numbers, and hence they follow the power laws. That is, to multiply, the exponents of the two numbers are added, and to divide, the exponents are subtracted.

```
mul :: Dim -> Dim -> Dim
(Dim le1 ma1 ti1 cu1 te1 su1 lu1) `mul` (Dim le2 ma2 ti2 cu2 te2 su2 lu2) =
  Dim (le1+le2) (ma1+ma2) (ti1+ti2) (cu1+cu2) (te1+te2) (su1+su2) (lu1+lu2)
```

**Exercise** Implement a function for dividing two dimensions.

## ▼ Solution

```
div :: Dim -> Dim -> Dim
(Dim le1 ma1 ti1 cu1 te1 su1 lu1) `div` (Dim le2 ma2 ti2 cu2 te2 su2 lu2) =
  Dim (le1-le2) (ma1-ma2) (ti1-ti2) (cu1-cu2) (te1-te2) (su1-su2) (lu1-lu2)
```

It's now possible to construct dimensions in the following fashion.

```
velocity' = length `div` time
area'     = length `mul` length
force     = mass   `mul` acceleration
momentum  = force  `mul` time
```

A dimension we so far haven't mentioned is the *scalar*, which shows up when working with, for example, coefficients of friction. It's dimensionless because it arises from division of two equal dimensions. The case of coefficients of friction looks like

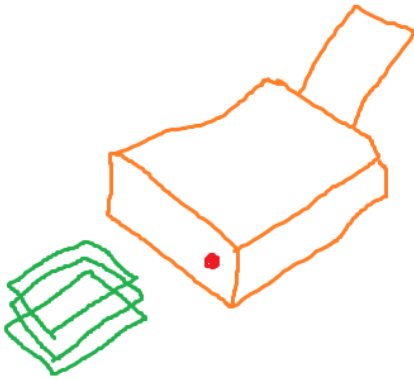
$$F_{friction} = \mu * F_{normal} \iff \mu = \frac{F_{friction}}{F_{normal}}$$

**Exercise** Create two values, which represent the scalar. They should of course have the same value, but be created in two different ways. One by writing the exponents explicitly. One by dividing two equal dimensions.

### ▼ Solution

```
one = Dim 0 0 0 0 0 0 0
one' = force `div` force
```

## Pretty-printer



The purpose of value-level dimensions is to be able to print 'em nicely. The pretty printer should be function with the type

```
showDim :: Dim -> String
```

meaning it shows a dimension as a string. But how to actually implement this is not the interesting part of this tutorial. Hence we skip it.

We use `showDim` to make `Dim` an instance of `Show`

```
instance Show Dim where
  show = showDim
```

Now dimensions are printed in a quite pretty way in GHCi

```
ghci> momentum
kg*m/s
```

Note that the SI-unit of the dimensions is used for printing.

The result from this section is the ability to multiply, divide and print dimensions. The next step is to test these operations and see if they actually work.

[src:  
[Dimensions/ValueLevel.lhs](#)]

Previous:  
[Introduction](#)

[Table of  
contents](#)

Next: [Testing of value-level  
dimensions](#)

© Björn Werner, Erik Sjöström, Johan Johansson, Oskar Lundström (2018), GPL

# Learn You a **Physics** for Great Good!

>>> WORK IN PROGRESS <<<

## Dimensions / Testing of value-level dimensions

[src: [Dimensions/ValueLevel/Test.lhs](#)]

Previous: [Value-level](#)

[Table of contents](#)

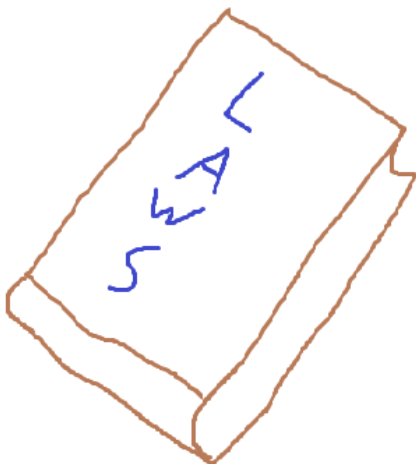
Next: [Type-level dimensions](#)

## Testing of value-level dimensions

```
module Dimensions.ValueLevel.Test
  ( runTests
  ) where

import Prelude hiding (length, div)
import Test.QuickCheck
import Data.List

import Dimensions.ValueLevel
```



For operations on dimensions, there are a number of laws which should hold. We will here test that the value-level dimensions obey them. One way is to use QuickCheck, which produces lots o' random test cases.

### Generating arbitrary dimensions

The first thing one needs in order to use QuickCheck is an instance of Arbitrary for the data type to be used in the

tests. In this case it's Dim.

An arbitrary example of an Arbitrary instance (get it?) could look like

```
data IntPair = IntPair (Int, Int)

genIntPair :: Gen IntPair
genIntPair = do
  first <- arbitrary
  second <- arbitrary
  return $ IntPair (first, second)

instance Arbitrary IntPair where
  arbitrary = genIntPair
```

**Exercise** Now try to implement an Arbitrary instance of Dim.

### ▼ Solution

Here's one way to do it.

```
genDim :: Gen Dim
genDim = do
  le <- arbitrary
  ma <- arbitrary
  ti <- arbitrary
  cu <- arbitrary
  te <- arbitrary
  su <- arbitrary
  lu <- arbitrary
  return (Dim le ma ti cu te su lu)

instance Arbitrary Dim where
  arbitrary = genDim
```

## Properties for operations on dimensions

Since dimensions are treated just like regular numbers when it comes to multiplication and division, the laws which ought to hold should be pretty clear. It's the "obvious" laws such as commutativity and so on.



The laws to test are

- Multiplication is commutative
- Multiplication is associative
- one is a unit for multiplication
- Multiplication and division cancel each other out
- Dividing by one does nothing
- Dividing by a division brings up the lowest denominator

$$\frac{x}{\frac{x}{y}} = y$$

- Multiplication by  $x$  is the same as dividing by the inverse of  $x$ .

The implementation of the first law looks like

```
-- Property: multiplication is commutative
prop_mulCommutative :: Dim -> Dim -> Bool
prop_mulCommutative d1 d2 = d1 `mul` d2 == d2 `mul` d1
```

**Exercise.** Implement the rest.

### ▼ Solution

Here's what the rest could look like.

```
-- Property: multiplication is associative
prop_mulAssociative :: Dim -> Dim -> Dim -> Bool
prop_mulAssociative d1 d2 d3 = d1 `mul` (d2 `mul` d3) ==
  (d1 `mul` d2) `mul` d3

-- Property: `one` is a unit for multiplication
prop_mulOneUnit :: Dim -> Bool
prop_mulOneUnit d = d == one `mul` d

-- Property: multiplication and division cancel each other out
prop_mulDivCancel :: Dim -> Dim -> Bool
prop_mulDivCancel d1 d2 = (d1 `mul` d2) `div` d1 == d2
```

```

-- Property: dividing by `one` does nothing
prop_divOne :: Dim -> Bool
prop_divOne d = d `div` one == d

-- Property: dividing by a division brings up the lowest denominator
prop_divTwice :: Dim -> Dim -> Bool
prop_divTwice d1 d2 = d1 `div` (d1 `div` d2) == d2

-- Property: multiplication same as division by inverse
prop_mulDivInv :: Dim -> Dim -> Bool
prop_mulDivInv d1 d2 = d1 `mul` d2 ==
  d1 `div` (one `div` d2)

```

We should also test the pretty-printer. But just like how that function itself is implemented isn't interesting, neither is the code testing it. We therefore leave it out.

From this module, we export a function `runTests`. That function runs all the tests implemented here and is used with `Stack`.

[src: [Dimensions/ValueLevel/Test.lhs](#)      Previous: [Value-level dimensions](#)      [Table of contents](#)      Next: [Type-level dimensions](#)]

© Björn Werner, Erik Sjöström, Johan Johansson, Oskar Lundström (2018), GPL

# Learn You a **Physics** for Great Good!

>>> WORK IN PROGRESS <<<

## Dimensions / Type-level dimensions

[src: [Dimensions/TypeLevel.lhs](#)] Previous: [Testing of value-level dimensions](#) [Table of contents](#) Next: [Quantities](#)

## Type-level dimensions

```
module Dimensions.TypeLevel
  ( Dim(..)
  , Mul
  , Div
  , Length
  , Mass
  , Time
  , Current
  , Temperature
  , Substance
  , Luminosity
  , One
  ) where
```

We will now implement *type-level* dimensions. What is type-level? Programs (in Haskell) normally operate on (e.g. add) values (e.g. 1 and 2). This is on *value-level*. Now we'll do the same thing but on *type-level*, that is, perform operations on types.

What's the purpose of type-level dimensions? It's so we'll notice as soon as compile-time if we've written something incorrect. E.g. adding a length and an area is not allowed since they have different dimensions.



This implementation is very similar to the value-level one. It would be possible to only have one implementation by using `Data.Proxy`. But it would be trickier. This way is lengthier but easier to understand.

To be able to do type-level programming, we'll need a nice stash of GHC-extensions.

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE UndecidableInstances #-}
{-# LANGUAGE TypeOperators #-}
```

See the end of the next chapter to read what they do.

We'll need to be able to operate on integers on the type-level. Instead of implementing it ourselves, we will just import the machinery so we can focus on the physics-part.

```
import Numeric.NumType.DK.Integers
```

We make a *kind* for dimensions, just like we in the previous section made *type* for dimensions. On value-level we made a *type* with *values*. Now we make a *kind* with *types*. The meaning is exactly the same, except we have moved “one step up”.

```

data Dim = Dim TypeInt -- Length
                TypeInt -- Mass
                TypeInt -- Time
                TypeInt -- Current
                TypeInt -- Temperature
                TypeInt -- Substance
                TypeInt -- Luminosity

```

But `data Dim = ...` looks awfully similar to a regular data type! That's correct. But with the GHC-extension `DataKinds` this will, apart from creating a regular data type, **also** create a *kind*. Perhaps a less confusing syntax would've been `kind Dim = ...`. The above definition can be seen as the two following definitions.

```

-- LHS: Type
-- RHS: Value
data Dim = Dim TypeInt
                TypeInt
                TypeInt
                TypeInt
                TypeInt
                TypeInt
                TypeInt

```

```

-- LHS: Kind
-- RHS: Type
kind Dim = Dim TypeInt
                TypeInt
                TypeInt
                TypeInt
                TypeInt
                TypeInt
                TypeInt

```

Thanks to the `Dim`-kind we can force certain types in functions to be of this kind.

This may sound confusing, but the point of this will become clear over time. Let's show some example *types* of the `Dim`-kind.

```

type Length      = 'Dim Pos1 Zero Zero Zero Zero Zero Zero
type Mass        = 'Dim Zero Pos1 Zero Zero Zero Zero Zero
type Time        = 'Dim Zero Zero Pos1 Zero Zero Zero Zero
type Current     = 'Dim Zero Zero Zero Pos1 Zero Zero Zero
type Temperature = 'Dim Zero Zero Zero Zero Pos1 Zero Zero

```

```
type Substance = 'Dim Zero Zero Zero Zero Zero Pos1 Zero
type Luminosity = 'Dim Zero Zero Zero Zero Zero Zero Pos1
```

'Dim is used to distinguish between the *type* Dim (left-hand-side of the data Dim definition) and the *type constructor* Dim (right-hand-side of the data Dim definition, with DataKinds-perspective). 'Dim refers to the type constructor. Both are created when using DataKinds.

Pos1, Neg1 and so on corresponds to 1 and -1 in the imported package, which operates on type-level integers.

**Exercise** Create types for velocity, acceleration and the scalar.

### ▼ Solution

```
type Velocity      = 'Dim Pos1 Zero Neg1 Zero Zero Zero Zero
type Acceleration = 'Dim Pos1 Zero Neg2 Zero Zero Zero Zero

type One = 'Dim Zero Zero Zero Zero Zero Zero Zero
```

## Multiplication and division

Let's implement multiplication and division on the type-level. After such an operation a new dimension is created. And from the previous section we already know what the dimension should look like. To translate to Haskell-language: "after such an operation a new *type* is created". How does one implement that? With type family! A type family can easiest be thought of as a function on the type-level.

```
type family Mul (d1 :: Dim) (d2 :: Dim) where
  Mul ('Dim le1 ma1 ti1 cu1 te1 su1 lu1)
      ('Dim le2 ma2 ti2 cu2 te2 su2 lu2) =
      'Dim (le1+le2) (ma1+ma2) (ti1+ti2) (cu1+cu2)
          (te1+te2) (su1+su2) (lu1+lu2)
```

- type family means it's a function on type-level.
- Mul is the name of the function.
- d1 :: Dim is read as "the *type* d1 has *kind* Dim".

**Exercise** As you would suspect, division is very similar, so why don't you try 'n implement it yourself?

### ▼ Solution

```
type family Div (d1 :: Dim) (d2 :: Dim) where
  Div ('Dim le1 ma1 ti1 cu1 te1 su1 lu1)
    ('Dim le2 ma2 ti2 cu2 te2 su2 lu2) =
    'Dim (le1-le2) (ma1-ma2) (ti1-ti2) (cu1-cu2)
      (te1-te2) (su1-su2) (lu1-lu2)
```

**Exercise** Implement a type-level function for raising a dimension to the power of some integer.

### ▼ Solution

```
type family Power (d :: Dim) (n :: TypeInt) where
  Power ('Dim le ma ti cu te su lu) n =
    'Dim (le*n) (ma*n) (ti*n) (cu*n) (te*n) (su*n) (lu*n)
```

Now types for dimensions can be created by combining existing types, much like we did for values in the previous chapter.

**Exercise** Create types for velocity, area, force and impulse.

### ▼ Solution

```
type Velocity' = Length `Div` Time
type Area      = Length `Mul` Length
type Force     = Mass   `Mul` Length
type Impulse   = Force  `Mul` Time
```

Perhaps not very exciting so far. But just wait 'til we create a data type for quantities. Then the strengths of type-level dimensions will be clearer.

[src:  
[Dimensions/TypeLevel.lhs](#)]

Previous: [Testing of value-level  
dimensions](#)

[Table of  
contents](#)

Next:  
[Quantities](#)

© Björn Werner, Erik Sjöström, Johan Johansson, Oskar Lundström (2018), GPL



# Learn You a **Physics** for Great Good!

>>> WORK IN PROGRESS <<<

## Dimensions / Quantities

[src:  
[Dimensions/Quantity.lhs](#)]

Previous: [Type-level  
dimensions](#)

[Table of  
contents](#)

Next: [Testing of  
Quantities](#)

## Quantities

```
module Dimensions.Quantity
  ( Quantity
  , length, mass, time, current, temperature, substance, luminosity, one
  , velocity, acceleration, force, momentum
  , meter, kilogram, second, ampere, kelvin, mole, candela, unitless
  , (~=)
  , isZero
  , (#)
  , (+#), (-#), (*#), (/#)
  , sinq, cosq, asinq, acosq, atanq, expq, logq
  ) where

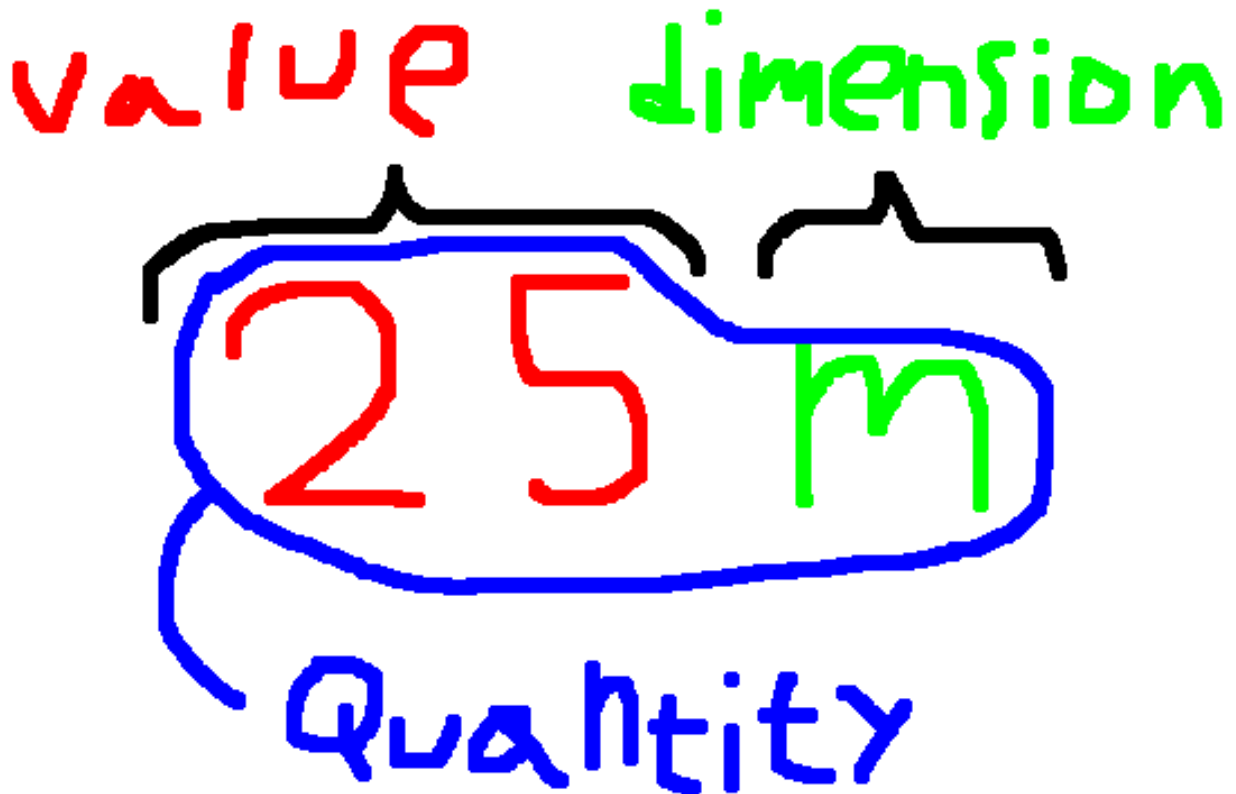
import qualified Dimensions.ValueLevel as V
import           Dimensions.TypeLevel  as T
import           Prelude                as P hiding (length)
```

We'll now create a data type for quantities and combine dimensions on value-level and type-level. Just as before, a bunch of GHC-extensions are necessary.

```
{-# LANGUAGE UndecidableInstances #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE GADTs #-}
```

```
{-# LANGUAGE DataKinds #-}  
{-# LANGUAGE TypeOperators #-}  
{-# LANGUAGE KindSignatures #-}
```

So what exactly does a physical “quantity” contain? Have a look at this picture



This is what quantities look like in physics calculations. They have a *numerical value* and a *dimension* (which is often given by instead writing its *unit*). The whole thing combined is the *quantity*. This combined thing is what we want to have a data type for in Haskell.

It's evident the data type should have a numerical value and a dimension. But so far we have created two dimensions! Which should we use? Both! The value-level dimension of the quantity is used to print it nicely. The type-level dimension of the quantity is to get some type-safety. Just like this

$$25m + 7kg = \dots$$

should upset you, this (in pseudo-Haskell)

```
ghci> let q1 = 25 m -- A value of the quantity type  
ghci> let q2 = 7 kg -- Another value of the quantity type  
ghci> q1 + q2  
...
```

should upset the compiler!

We can't cover all everything at once, but we guarantee you that by the end of this chapter, you'll know exactly how the above ideas are actually implemented.

Let's get on to the actual data type declaration.

```
data Quantity (d :: T.Dim) (v :: *) where
  ValQuantity :: V.Dim -> v -> Quantity d v
```

That was sure a mouthful! Let's break it down. `data Quantity (d :: T.Dim) (v :: *)` creates the *type constructor* `Quantity`. A type constructor takes types to create another type. In this case, the type constructor `Quantity` takes a type `d` of *kind* `T.Dim` and a type `v` of *kind* `*` to create the type `Quantity d v`. Let's see it in action

```
type ExampleType = Quantity T.Length Double
```

`ExampleType` is the type representing quantities of the physical dimension length and where the numerical value is of the type `Double`.

How do we create values of this type? That's where the second row comes in! `ValQuantity` is a *value constructor*, which means it takes values to create another value. In this case, the value constructor `ValQuantity` takes a value of *type* `V.Dim` and `v`. Let's see this one in action as well

```
exampleValue = ValQuantity V.length 25.0
```

`exampleValue` is a value for the quantity representing a length with the numerical value `25.0`.

We can combine the two and write this

```
exampleValue :: ExampleType
```

or

```
exampleValue :: Quantity T.Length Double
exampleValue = ValQuantity V.length 25.0
```

to get the full picture. So, here we are, with a way to create quantity values (`ValQuantity`) of the type `Quantity t1 t2` where `t1` is the type-level dimension of the quantity and `t2` the type of the numerical value. If you still find it confusing, don't worry! Over the course of this chapter, it'll become more concrete as we work more with `Quantity`.

Also note that the type-level dimension and value-level dimension are intended to match when used with `Quantity`! So far nothing enforces this. We'll tackle this problem later.

**Exercise** create `Quantity` values for 2.5 meters, 6.7 seconds and 9 mol. Recall that the three mentioned SI-units here, and all SI-units in this tutorial in general, have a one-to-one correspondence with their respective dimension.

### ▼ Solution

```
theDistance :: Quantity T.Length Double
theDistance = ValQuantity V.length 2.5
```

```
theTime :: Quantity T.Time Double
theTime = ValQuantity V.time 6.7
```

```
theAmountOfSubstance :: Quantity T.Substance Integer
theAmountOfSubstance = ValQuantity V.substance 9
```

**Exercise** Create a data type which has two type-level dimensions, always use `Rational` as the value-holding type (the role of `v`) but which has no value-level dimensions. It should have three numerical values. Create some values of that type.

### ▼ Solution

```
data Quantity' (d1 :: T.Dim) (d2 :: T.Dim) where
  ValQuantity' :: Rational -> Rational -> Rational -> Quantity' d1 d2

val1 :: Quantity' T.Luminosity T.Mass
val1 = ValQuantity' 8 7 3
```

```
val2 :: Quantity' T.Substance T.Temperature
val2 = ValQuantity' 2 3 1
```

## Pretty-printer

Let's do a pretty-printer for quantities. Most of the work is already done by the value-level dimensions.

```
showQuantity :: (Show v) => Quantity d v -> String
showQuantity (ValQuantity d v) = show v ++ " " ++ show d
```

```
instance (Show v) => Show (Quantity d v) where
  show = showQuantity
```

**Exercise** In a previous exercise, you created some example values of the `Quantity` type. Write them in GHCi and see how they look.

### ▼ Solution

```
ghci> theDistance
2.5 m
ghci> theTime
6.7 s
ghci> theAmountOfSubstance
9 mol
```

Pretty, huh?

## A taste of typos

We'll implement all arithmetic operations on `Quantity`, but for now, to get a taste of types, we show here addition and multiplication and some examples of values of type `Quantity`.

```

quantityAdd :: (Num v) => Quantity d v ->
              Quantity d v ->
              Quantity d v
quantityAdd (ValQuantity d v1) (ValQuantity _ v2) = ValQuantity d (v1+v2)

```

The type is interpreted as follows: two values of type `Quantity d v` is the input, where `d` is the type-level dimension. The output is also a value of type `Quantity d v`.

The type of the function forces the inputs to have the same dimensions. For this reason, the dimension on value-level doesn't matter on one of the arguments, because they will be the same. As was already mentioned when the `Quantity` type was created, it's possible to create values where the dimensions on value-level and type-level don't match. Just like then, we ignore this problem for now and fix it later!

Multiplication is implemented as

```

quantityMul :: (Num v) => Quantity d1 v ->
              Quantity d2 v ->
              Quantity (d1 `Mul` d2) v
quantityMul (ValQuantity d1 v1) (ValQuantity d2 v2) =
  ValQuantity (d1 `V.mul` d2) (v1*v2)

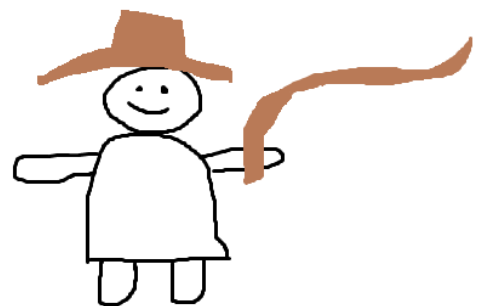
```

The type has the following interpretation: two values of type `Quantity dx v` is input, where `dx` are two types representing (potentially different) dimensions. As output, a value of type `Quantity` is returned. The type in the `Quantity` will be the type that is the product of the two dimensions in.

**Exercise** Implement subtraction and division.

### ▼ Solution

Hold on cowboy! Not so fast. We'll come back later to subtraction and division, so check your solution then.



Now on to some example values and types.

```
width :: Quantity T.Length Double
```

```
width = ValQuantity V.length 0.5
```

```
height :: Quantity T.Length Double
```

```
height = ValQuantity V.length 0.3
```

```
type Area = Mul T.Length T.Length
```

The following example shows that during a multiplication, the types will change, as they should. The dimensions change not only on value-level but also on type-level.

```
area :: Quantity Area Double
```

```
area = quantityMul width height
```

```
ghci> width
```

```
0.5 m
```

```
ghci> :t width
```

```
width :: Quantity Length Double
```

```
ghci> area
```

```
0.15 m2
```

```
ghci> :i area
```

```
area :: Quantity Area Double
```

(Try out `:t` instead of `:i` on the last one and see what happens!)

The type-level dimensions are used below to enforce, at compile-time, that only allowed operations are attempted.

```
-- Doesn't even compile
```

```
weird = quantityAdd width area
```

If the dimensions had been value-level only, this error would go undetected until run-time.

**Exercise** What is the volume of a cuboid with sides  $1.2\text{ m}$ ,  $0.4\text{ m}$  and  $1.9\text{ m}$ ? Create values for the involved quantities. Use `Float` instead of `Double`.

### ▼ Solution

```
side1 :: Quantity Length Float
```

```
side1 = ValQuantity V.length 1.2
```

```

side2 :: Quantity Length Float
side2 = ValQuantity V.length 0.4

side3 :: Quantity Length Float
side3 = ValQuantity V.length 1.9

type Volume = Length `Mul` (Length `Mul` Length)

volume :: Quantity Volume Float
volume = quantityMul side1 (quantityMul side2 side3)

```

## Comparisons

It's useful to be able to compare quantities. Perhaps one wants to know which of two amounts of energy is the largest. But what's the largest of 1 J and 1 m? That's no meaningful comparison since the dimensions don't match. This behaviour is prevented by having type-level dimensions.

```

quantityEq :: (Eq v) => Quantity d v -> Quantity d v -> Bool
quantityEq (ValQuantity _ v1) (ValQuantity _ v2) = v1 == v2

```

```

instance (Eq v) => Eq (Quantity d v) where
  (==) = quantityEq

```

**Exercise** Make Quantity an instance of Ord.

### ▼ Solution

```

quantityCompare :: (Ord v) => Quantity d v ->
                    Quantity d v -> Ordering
quantityCompare (ValQuantity _ v1) (ValQuantity _ v2) =
  compare v1 v2

instance (Ord v) => Ord (Quantity d v) where
  compare = quantityCompare

```



We often use `Double` as the value holding type. Doing exact comparisons isn't always possible due to rounding errors. Therefore, we'll create a `~=` function for testing if two quantities are almost equal.

```
infixl 4 ~=  
(~=) :: Quantity d Double -> Quantity d Double -> Bool  
(ValQuantity _ v1) ~= (ValQuantity _ v2) = abs (v1-v2) < 0.001
```

Testing if a quantity is zero is something which might be a common operation. So we define it here.

```
isZero :: (Fractional v, Ord v) => Quantity d v -> Bool  
isZero (ValQuantity _ v) = abs v < 0.001
```

## Arithmetic on quantities

Let's implement the arithmetic operations on `Quantity`. Basically it's all about creating functions with the correct type-level dimensions.

```
infixl 6 +#  
(+#) :: (Num v) => Quantity d v -> Quantity d v ->  
      Quantity d v  
(+#) = quantityAdd
```

```
infixl 6 -#  
(-#) :: (Num v) => Quantity d v -> Quantity d v ->  
      Quantity d v  
(ValQuantity d v1) -# (ValQuantity _ v2) = ValQuantity d (v1-v2)
```

```
infixl 7 *#  
(*#) :: (Num v) => Quantity d1 v -> Quantity d2 v ->  
      Quantity (d1 `Mul` d2) v  
(*#) = quantityMul
```

```
infixl 7 /#  
(/#) :: (Fractional v) => Quantity d1 v ->  
      Quantity d2 v ->
```

```

Quantity (d1 `Div` d2) v
(ValQuantity d1 v1) /# (ValQuantity d2 v2) =
  ValQuantity (d1 `V.div` d2) (v1 / v2)

```

For all operations on quantities, one does the operation on the value and, in the case of multiplication and division, on the dimensions separately. For addition and subtraction the in-dimensions must be the same. Nothing then happens with the dimension.

How does one perform operations such as `sin` on a quantity with a *potential* dimension? The answer is that the quantity must be dimensionless, and with the dimension nothing happens.

Why is that, one might wonder. Every function can be written as a power series. For *sin* the series looks like

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots$$

The dimension of *every* term must be the same. The only way for that to be case is if the input  $x$  is dimensionless. Then so will the output.

The other functions can be written as similar power series and we'll see on those too that the input and output must be dimensionless.

```

sinq :: (Floating v) => Quantity One v -> Quantity One v
sinq (ValQuantity d1 v) = ValQuantity d1 (sin v)

```

```

cosq :: (Floating v) => Quantity One v -> Quantity One v
cosq (ValQuantity d1 v) = ValQuantity d1 (cos v)

```

We quickly realize a pattern, so let's generalize a bit.

```

qmap :: (a -> b) -> Quantity One a -> Quantity One b
qmap f (ValQuantity d1 v) = ValQuantity d1 (f v)

```

```

qmap' :: (a -> b) -> Quantity dim a -> Quantity dim b
qmap' f (ValQuantity d v) = ValQuantity d (f v)

```

```

qfold :: (a -> a -> b) -> Quantity dim a -> Quantity dim a -> Quantity dim b
qfold f (ValQuantity d v1) (ValQuantity _ v2) = ValQuantity d (f v1 v2)

```

```

sinq, cosq, asinq, acosq, atanq, expq, logq :: (Floating v) =>
  Quantity One v -> Quantity One v
sinq = qmap sin
cosq = qmap cos
asinq = qmap asin
acosq = qmap acos
atanq = qmap atan
expq = qmap exp
logq = qmap log

```

Why not make `Quantity` an instance of `Num`, `Fractional`, `Floating` and `Functor`? The reason is that the functions of those type classes have the following type

```
(*) :: (Num a) => a -> a -> a
```

which isn't compatible with `Quantity` since multiplication with `Quantity` has the following type

```
(*#) :: (Num v) => Quantity d1 v -> Quantity d2 v ->
  Quantity (d1 `Mul` d2) v
```

The input here may actually be of *different* types, and the output has a type depending on the types of the input. However, the *kind* of the inputs and output are the same, namely `Quantity`. We'll just have to live with not being able to make `Quantity` a `Num`-instance.

However, operations with only scalars (type `One`) has types compatible with `Num`.

**Exercise** `Quantity One` has compatible types. Make it an instance of `Num`, `Fractional`, `Floating` and `Functor`.

### ▼ Solution

```

instance (Num v) => Num (Quantity One v) where
  (+) = (+#)
  (-) = (-#)
  (*) = (*#)
  abs = qmap abs
  signum = qmap signum
  fromInteger n = ValQuantity V.one (fromInteger n)

```

```
instance (Fractional v) => Fractional (Quantity One v) where
  (/) = (/#)
  fromRational r = ValQuantity V.one (fromRational r)
```

```
instance (Floating v) => Floating (Quantity One v) where
  pi    = ValQuantity V.one pi
  exp   = expq
  log   = logq
  sin   = sinq
  cos   = cosq
  asin  = asinq
  acos  = acosq
  atan  = atanq
  sinh  = qmap sinh
  cosh  = qmap cosh
  asinh = qmap asinh
  acosh = qmap acosh
  atanh = qmap atanh
```

```
instance Functor (Quantity One) where
  fmap = qmap
```

## Syntactic sugar

In order to create a value representing a certain distance (5 metres, for example) one does the following

```
distance :: Quantity T.Length Double
distance = ValQuantity V.length 5.0
```

Writing that way each time is very clumsy. You can also do “dumb” things such as

```
distance :: Quantity T.Length Double
distance = ValQuantity V.time 5.0
```

with different dimensions on value-level and type-level.

To solve these two problems we’ll introduce some syntactic sugar. First some pre-made values for the 7 base dimensions and the scalar.

```
length :: (Num v) => Quantity Length v
length = ValQuantity V.length 1
```

```
mass :: (Num v) => Quantity Mass v
mass = ValQuantity V.mass 1
```

```
time :: (Num v) => Quantity Time v
time = ValQuantity V.time 1
```

**Exercise** Do the rest.

### ▼ Solution

```
current :: (Num v) => Quantity Current v
current = ValQuantity V.current 1
```

```
temperature :: (Num v) => Quantity Temperature v
temperature = ValQuantity V.temperature 1
```

```
substance :: (Num v) => Quantity Substance v
substance = ValQuantity V.substance 1
```

```
luminosity :: (Num v) => Quantity Luminosity v
luminosity = ValQuantity V.luminosity 1
```

```
one :: (Num v) => Quantity One v
one = ValQuantity V.one 1
```

And now the sugar.

```
(#) :: (Num v) => v -> Quantity d v -> Quantity d v
v # (ValQuantity d bv) = ValQuantity d (v*bv)
```

The intended usage of the function is the following

```
ghci> let myDistance = 5 # length
ghci> :t myDistance
```

```
t :: Num v => Quantity Length v
ghci> myDistance
5 m
```

To create a Quantity with a certain value (here 5) and a certain dimension (here length), you write as above and get the correct dimension on both value-level and type-level.

This way of writing in Haskell is similar to what you do in traditional physics.



Both mean that “now we get 5 pieces of the SI-unit meters”. This is signified in the implementation by  $b \cdot b_v$  where  $b_v$  stands for “base value”. The base value is the SI-unit, which is 1. Since “length” and “meter” are equivalent in our implementation, *length means “meter”*.

But let’s not stop there. It would be prettier if you could actually write meter instead of length. In fact, not much code is needed for this!

**Exercise** Make it so that one can write the SI-units instead of the base dimensions when one uses the sugar. Then show how to write 4 seconds.

▼ **Solution**

```
meter = length
kilogram = mass
second = time
ampere = current
kelvin = temperature
mole = substance
candela = luminosity
unitless = one
```



```
fourSeconds = 4 # second
```

That's nice and all, but we can actually take this way of thinking even further. The sugar-function `#` multiplies with the *base* value of a unit. Thanks to this, we can actually support more units than just the SI-units! (At least for input.)

**Exercise** Make it so that one can write units such as inch, foot and pound in the same way one can write the SI-units.

### ▼ Hint

`x # unit` multiplies `x`, how many pieces of the unit you want, with the base value of unit. So you want to think about how much of the corresponding SI-unit a unit corresponds to.

### ▼ Solution

We just need to create pre-made values for the units we want to support.

```
inch = 0.0254 # meter
foot = 0.3048 # meter
pound = 0.45359237 # kilogram
```

All right, so now we have tackled the problem of writing quantities easier than before. What about the second problem?

We want to maintain the invariant that the dimension on value-level and type-level always match. The pre-made values from above maintain it, and so do the arithmetic operations we previously created. Therefore, we only export those from this module! The user will have no choice but to use these constructs and hence the invariant will be maintained.

If the user needs other dimensions than the base dimensions (which it probably will), the following example shows how it's done.

```
ghci> let myLength = 5 # length
ghci> let myTime = 2 # time
ghci> let myVelocity = myLength /# myTime
ghci> myVelocity
2.5 m/s
```

New dimensions are created “on demand”. Furthermore

```
ghci> let velocity = length /# time
ghci> let otherVelocity = 188 # velocity
```

it's possible to use the sugar from before on user-defined dimensions.

Just for convenience sake we'll define a bunch of common composite dimensions. Erm, *you'll* define.

**Exercise** Define pre-made values for velocity, acceleration, force, momentum and energy.

### ▼ Solution

```
velocity      = length    /# time
acceleration  = velocity  /# time
force         = mass      *# acceleration
momentum      = force     *# time
energy        = force     *# length
```

## A physics example

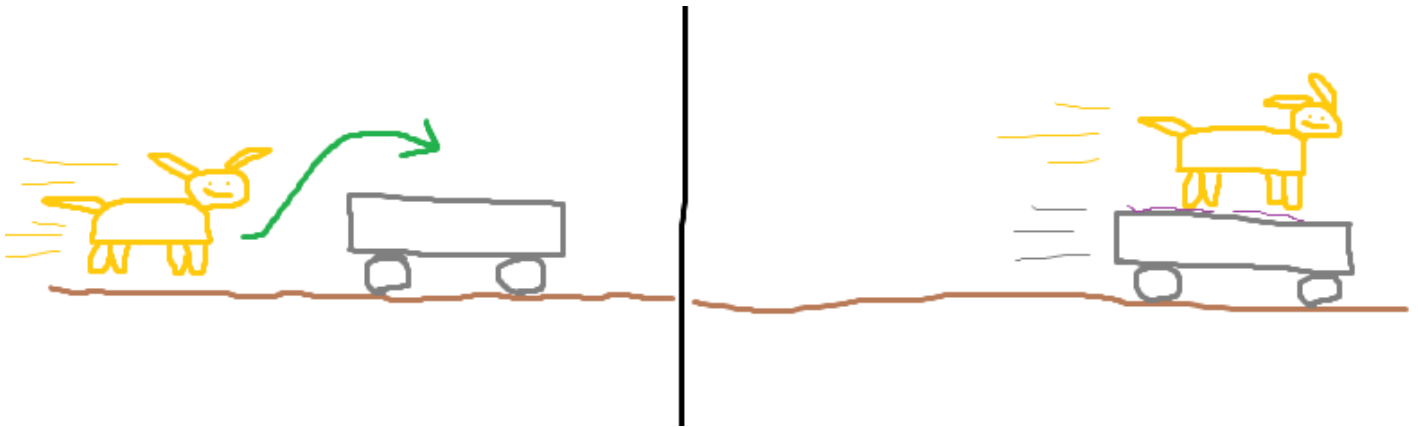


To conclude this chapter, we show an example on how to code an exercise and its solution in this domain specific language we've created for quantities.

The code comments show what GHCi prints for that line.

The exercise is "A dog runs, jumps and lands sliding on a carriage. The dog weighs  $40\text{ kg}$  and runs in  $8\text{ m/s}$ . The carriage weighs  $190\text{ kg}$ . The coefficient of friction between the dog's paws and the carriage is  $0.7$ ."

This is illustrated in the painting below.



a. Calculate the (shared) final velocity of the dog and the carriage.

```
mDog      = 40 # mass      -- 40 kg
viDog     = 8 # velocity  -- 8 m/s
mCarriage = 190 # mass    -- 190 kg
u         = 0.7 # one     -- 0.7
```

No external forces are acting on the dog and the carriage. Hence the momentum of the system is preserved.

```
piSystem = mDog *# viDog -- 320 kg*m/s
pfSystem = piSystem     -- 320 kg*m/s
```

In the end the whole system has a shared velocity of its shared mass.

```
mSystem = mDog +# mCarriage -- 230 kg
vfSystem = pfSystem /# mSystem -- 1.39 m/s
```

b. Calculate the force of friction acting on the dog.

```
fFriction = u *# fNormal      -- 275 kg*m/s^2
fNormal   = mDog *# g         -- 393 kg*m/s^2
g         = 9.82 # acceleration -- 9.82 m/s^2
```

c. For how long time does the dog slide on the carriage?

```
aDog = fFriction /# mDog -- 6.87 m/s^2
```

```
vDelta = mDog -# vfSystem
```

```
* Couldn't match type 'Numeric.NumType.DK.Integers.Neg1
    with 'Numeric.NumType.DK.Integers.Zero
...

```

Whoops! That's not a good operation. Luckily the compiler caught it.

```
vDelta = viDog -# vfSystem -- 6.60 m/s
tSlide = vDelta /# aDog    -- 0.96 s
```

## Conclusion

Okay, so you've read a whole lot of text by now. But in order to really learn anything, you need to practise with some additional exercises. Some suggestions are

- Implement first value-level dimensions, then type-level dimensions and last quantities by yourself, without looking too much at this text.
- Implement a power function.
- Implement a square-root function. The regular square-root function in Haskell uses `exp` and `log`, which only work on `Quantity One`. But taking the square-root of e.g. an area should be possible.
- Extending the `Quantity` data type here by adding support for prefixes and non-SI-units.
- Ditching the separate implementations of value-level and type-level, and only use one with `Data.Proxy`.

After reading this text and doing some exercises, we hope you've learnt

- The relation between dimensions, quantities and units.

- How dimensions...
  - ...restrict operations such as comparison to quantities of the same dimension.
  - ...change after operations such as multiplication.
  - ...can be implemented on the type-level in Haskell to enforce the two above at compile-time.
- The basics about kinds and type-level programming in Haskell.

## Further reading

The package [Dimensional](#) inspired a lot of what we did here. Our `Quantity` is like a “lite” version of `Dimensional`, which among other things, support different units and use `Data.Proxy`.

The type-level programming in this text was done with the help of the tutorial [Basic Type Level Programming in Haskell](#). If you want to know more about kinds and type-level programming, it's a very good starting point.

[src:  
[Dimensions/Quantity.lhs](#)]

Previous: [Type-level  
dimensions](#)

[Table of  
contents](#)

Next: [Testing of  
Quantities](#)

© Björn Werner, Erik Sjöström, Johan Johansson, Oskar Lundström (2018), GPL

# Learn You a **Physics** for Great Good!

>>> WORK IN PROGRESS <<<

## Dimensions / Testing of Quantities

[src: [Dimensions/Quantity/Test.lhs](#)] Previous: [Quantities](#) [Table of contents](#) Next: [Usage](#)

### Testing of Quantity

```
module Dimensions.Quantity.Test
( runTests
)
where

{-# LANGUAGE DataKinds #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE UndecidableInstances #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE TypeSynonymInstances #-}

import Prelude hiding (length, div)
import Test.QuickCheck

import Dimensions.TypeLevel
import Dimensions.Quantity
```

Quantity consists of a numerical type such as `Double` and a value-level dimension. Let's assume Haskell's implementation of numbers abide the laws. We also know the value-

level dimensions abide the laws since since we tested them in a previous section. So what's there to test now? Well, it's still useful to test the top-level construct. And error may arise when combining different parts.

So let's test *some* things. Due to the type-level dimensions it's impossible to easily and readably design test for all combinations of dimensions. Hence it'll have to suffice with only some combinations.

## Generating arbitrary quantities

First we need an Arbitrary instance for Quantity d val. For d we'll mostly use One and for val we'll exclusively use Double.

```
type Q d = Quantity d Double
```

A generator for an arbitrary dimension.

```
genQuantity :: Quantity d Double -> Gen (Q d)
genQuantity quantity = do
  value <- arbitrary
  return (value # quantity)
```

And now we make Arbitrary instances of arbitrary selected dimensions in the Quantities.

```
instance Arbitrary (Q One) where
  arbitrary = genQuantity one
```

```
instance Arbitrary (Q Length) where
  arbitrary = genQuantity length
```

```
instance Arbitrary (Q Mass) where
  arbitrary = genQuantity mass
```

```
instance Arbitrary (Q Time) where
  arbitrary = genQuantity time
```

## Testing arithmetic properties

On regular numbers, and hence too on quantities with their dimensions, a bunch of properties should hold. The things we test here are

- Addition commutative
- Addition associative
- Zero is identity for addition
- Multiplication commutative
- Multiplication associative
- One is identity for multiplication
- Addition distributes over multiplication
- Subtraction and addition cancel each other out
- Division and multiplication cancel each other out
- Pythagorean trigonometric identity

Let's start!

We could write the type signatures in a general way like

```
prop_addCom :: Q d -> Q d -> Bool
```

But we won't do that since QuickCheck needs concrete types in order to work. So we would have to do a bunch of specialization anyway. And even if we begin with a general signature, we can't cover all cases since there are infinitely many dimensions.

Instead we'll pick some arbitrary dimensions that have an Arbitrary instance.

```
-- a + b = b + a
prop_addCom :: Q Length -> Q Length -> Bool
prop_addCom a b = (a +# b) ~= (b +# a)

-- a + (b + c) = (a + b) + c
prop_addAss :: Q Mass -> Q Mass -> Q Mass -> Bool
prop_addAss a b c = a +# (b +# c) ~= (a +# b) +# c

-- 0 + a = a
prop_addId :: Q Time -> Bool
prop_addId a = zero +# a ~= a
  where
    zero = 0 # a
```

```

-- a * b = b * a
prop_mulCom :: Q Length -> Q Mass -> Bool
prop_mulCom a b = a *# b ~== b *# a

-- a * (b * c) = (a * b) * c
prop_mulAss :: Q Time -> Q Length -> Q Mass -> Bool
prop_mulAss a b c = a *# (b *# c) ~==
                    (a *# b) *# c

-- 1 * a = a
prop_mulId :: Q Time -> Bool
prop_mulId a = (1 # one) *# a ~== a

-- a * (b + c) = a * b + a * c
prop_addDistOverMul :: Q Length -> Q Mass -> Q Mass -> Bool
prop_addDistOverMul a b c = a *# (b +# c) ~==
                             a *# b +# a *# c

-- (a + b) - b = a
prop_addSubCancel :: Q Length -> Q Length -> Bool
prop_addSubCancel a b = (a +# b) -# b ~== a

-- (a * b) / b = a
prop_mulDivCancel :: Q Time -> Q Length -> Property
prop_mulDivCancel a b = not (isZero b) ==>
    (a *# b) /# b ~== a

-- sin a * sin a + cos a * cos a = 1
prop_pythagoranIdentity :: Q One -> Bool
prop_pythagoranIdentity a = sinq a *# sinq a +#
                             cosq a *# cosq a ~== (1 # one)

```

[src: [Dimensions/Quantity/Test.lhs](#)]    Previous: [Quantities](#)    [Table of contents](#)    Next: [Usage](#)

© Björn Werner, Erik Sjöström, Johan Johansson, Oskar Lundström (2018), GPL

# Learn You a **Physics** for Great Good!

>>> WORK IN PROGRESS <<<

## Dimensions / Usage

[src:  
[Dimensions/Usage.lhs](#)]

Previous: [Testing of  
Quantities](#)

[Table of  
contents](#)

Next: [Single particle  
mechanics](#)

## Usage

```
module Dimensions.Usage
(
)
where
```

In this module we'll show how to use value-level dimensions, type-level dimensions and Quantity in an “actual” program. Let's first use this fancy GHC-extension

```
{-# LANGUAGE TypeOperators #-}
```

and then import the things we have created.

```
import Dimensions.TypeLevel
import Dimensions.Quantity
import Prelude hiding (length)
```

We don't need to import `Dimensions.ValueLevel` since all its features are available by using the `Quantity` data type. The same thing could be said about `Dimensions.TypeLevel`. So why do we import it? It's so we can write explicit type signatures, the trick of trade in Haskell programming.



# Values and types

Let's create a length, time and mass, in the way hinted in the previous chapter.

```
myLength = 10.0 # length
myTime   = 20.0 # time
myMass   = 30.0 # mass
```

Okay, let's check out their type and value in GHCi.

```
ghci> myLength
10.0 m
ghci> :t myLength
myLength :: Quantity Length Double
ghci> myTime
20.0 s
ghci> :t myTime
myTime :: Quantity Time Double
```

The pretty printing works thanks to dimensions on *value-level* and the Haskell-types thanks to the *type-level* dimensions. Recall that the value-level dimensions just is a pretty simple “average” data type.

Let's write some type signatures on those values (of type Quantity).

```
myLength :: Quantity Length Double
myLength = 10.0 # length
myTime   :: Quantity Time Double
myTime   = 20.0 # time
myMass   :: Quantity Mass Double
myMass   = 30.0 # mass
```

Neat! Length, Time and Mass are type-level dimensions. Therefore, Length *is* a type, just like String is a type.

Remember that the sugary style of writing `10.0 # length` is a shorthand for doing

```
myLength :: Quantity Length Double
myLength = Quantity length' 10.0
```

where `length'` refers to `length` in `Dimensions.ValueLevel`.

The reasons for not allowing this “raw” way of creating the value is to maintain the invariant of having the same value-level and type-level dimension. With the # function, the are *forced* by it to be the same. The following try to work around it is not possible.

```
myWeird :: Quantity Length Double
myWeird = 10.0 # time

* Couldn't match type 'Numeric.NumType.DK.Integers.Zero
      with 'Numeric.NumType.DK.Integers.Pos1
  Expected type: Quantity Length Double
  Actual type: Quantity Time Double
* In the expression: 10.0 # time
  In an equation for `myWeird': myWeird = 10.0 # time
```

## Custom values and types

Creating values of a dimensions *not* exported is also possible. Let’s take energy as an example.

```
energy = force *# length
```

energy can now be used just as length and so on.

```
energyToBoilMyGlassOfWater = 12 # energy
```

In order to write explicit type signatures, we can do like this

```
type Velocity = Length `Div` Time
type Acceleration = Velocity `Div` Time
type Force = Acceleration `Mul` Mass
type Energy = Force `Mul` Length
```

Note we had to introduce some helper types since, unlike the pre-made Quantity values, not as many were created for types.

Now energy can be defined and used like

```
energy :: Quantity Energy Double
energy = force *# length
```

```
energyToBoilMyGlassOfWater :: Quantity Energy Double
energyToBoilMyGlassOfWater = 12 # energy
```

## Some functions

Let's create a function operating on quantities. It should calculate the area of a rectangle given the lengths of its sides. The type should hence look like

```
areaOfRectangle :: Quantity Length Double -> Quantity Length Double ->
                Quantity (Length `Mul` Length) Double
```

And the actual function like

```
areaOfRectangle width height = width *# height
```

The values (width and height) can't be pattern matched on. This is so we can maintain the invariant we keep nagging about. The side effect is that only the comparative and arithmetic operations exported by `Dimensions.Quantity` can be used.

To wrap up, let's create a function to calculate the kinetic energy à la

$$E_k = \frac{m * v^2}{2}$$

```
kinecticEnergy :: Quantity Mass Double ->
                Quantity Velocity Double ->
                Quantity Energy Double
```

```
kinecticEnergy m v = m *# v *# v /# two
```

**where**

```
two = 2.0 # one
```

[src: [Dimensions/Usage.lhs](#)]

Previous: [Testing of Quantities](#)

[Table of contents](#)

Next: [Single particle mechanics](#)

# Learn You a **Physics** for Great Good!

## >>> WORK IN PROGRESS <<<

### Newtonian Mechanics / Single particle mechanics

[src: [NewtonianMechanics/SingleParticle.lhs](#)] Previous: [Usage](#) [Table of contents](#) Next: [Teeter](#)

```
module NewtonianMechanics.SingleParticle where
```

```
%< import Calculus.SyntaxTree
```

```
import Test.QuickCheck
```

Laws:

- A body remains at rest or in uniform motion unless acted upon by a force.
- If two bodies exert forces on each other, these forces are equal in magnitude and opposite in direction.

We will begin our journey into classical mechanics by studying point particles. A point particle has a mass and it's position is given as vector in three dimensions. Of course it could exist in any number of dimensions but we'll stay in three dimensions since it is more intuitive and easier to understand.

The components of the vector are functions over time that gives the particles position in each dimension, x, y, and z. Since we've already defined vectors and mathematical functions in previous chapters we won't spend any time on them here and instead just import those two modules.

```
import Calculus.FuncExpr
import Calculus.DifferentialCalc -- Maybe remove
import Calculus.IntegralCalc -- Maybe remove
import Vector.Vector as V
```

The mass of a particle is just a scalar value so we'll model it using *FunExpr*.

```
type Mass = FunExpr
```

We combine the constructor for vectors in three dimensions with the function expressions defined in the chapter on mathematical analysis. We'll call this new type `VectorE` to signify that it's a vector of expressions.

```
type VectorE = Vector3 FunExpr
```

Now we are ready to define what the data type for a particle is. As we previously stated a point particle has a mass, and a position given as a vector of function expressions. So our data type is simply:

```
data Particle = P { pos  :: VectorE -- Position as a function of time, unit m
                  , mass :: Mass   -- Mass, unit kg
                  } deriving Show
```

So now we can create our particles! Let's try it out!

```
particle :: Particle
particle = P (V3 (3 * Id * Id) (2 * Id) 1) 3
```

Let's see what happens when we run this in the interpreter.

```
ghci > particle
P {pos = (((3 * id) * id) x, (2 * id) y, 1 z), mass = 3}
```

We've created our first particle! And as we can see from the print out it's accelerating by  $3t^2$  in the x-dimension, has a constant velocity of  $2t$  in the y-dimension, is positioned at 1 in the z-dimension, and has a mass of 3.

## Velocity & Acceleration

Velocity is defined as the derivative of the position with respect to time. More formally, ( $\vec{r}$  denotes the position vector):

$$\vec{v} = \frac{d\vec{r}}{dt}$$

And since the position of our particles are given as vectors we'll do the derivation component-wise. We need not worry about the details of the derivation at all since this is all taken care of by the Calculus module, all we need to do is use the constructor `D` for the derivative and apply it to each of the components of the vector. The business of applying something to each component of a vector has also already been taken care of! This was the point of `vmap` to map a function over the components of the vector. So if we combine them we get a rather elegant way of computing the velocity of a particle.

```
velocity :: Particle -> VectorE
velocity = vmap D . pos
```

We can try this out in the interpreter with our newly created particle.

```
ghci > velocity particle
((D ((3 * id) * id)) x, (D (2 * id)) y, (D 1) z)
```

Not very readable but at least we can see that it correctly maps the derivative over the components of the vector.

*Acceleration* is defined as the derivative of the velocity with respect to time, or the second derivative of the position. More formally:

$$\vec{a} = \frac{d\vec{v}}{dt} = \frac{d^2\vec{r}}{dt^2}$$

**Exercise** Try to figure out how to define the function for calculating the acceleration of a particle.

### ▼ Solution

We already know how to get the velocity of a particle, so the only step we need to take is to take the derivative of the velocity.

```
acceleration :: Particle -> VectorE
acceleration = vmap D . velocity
```

Which is the same as:

```
acceleration :: Particle -> VectorE
acceleration = vmap D . vmap D . pos
```

## ▼ Trivia

Those of you familiar with functor laws will probably see that the code for calculating the acceleration could also be written as:

```
acceleration :: Particle -> VectorE
acceleration = vmap (D . D) . pos
```

## Forces & Newton's second law

Newton's second law states that

A body acted upon by a force moves in such a manner that the time rate of change of the momentum equals the force.

This law expresses the relationship between force and momentum and is mathematically defined as follows:

$$\vec{F} = \frac{d\vec{p}}{dt} = \frac{d(m \cdot \vec{v})}{dt}$$

Force is a vector so let's create a type synonym to make this clearer.

```
type Force = VectorE
```

The quantity  $m \cdot \vec{v}$  is what we mean when we say momentum. So the law states that the net force on a particle is equal to the rate of change of the momentum with respect to time. And since the definition of acceleration is  $\vec{a} = \frac{d\vec{v}}{dt}$  we can write this law in a more familiar form, namely:

$$\vec{F} = m \cdot \vec{a}$$

And thus if the particle is accelerating we can calculate the force that must be acting on it, in code this would be:

```

force :: Particle -> Force
force p = vmap (* m) a
  where
    m = mass p
    a = acceleration p

```

Where the acceleration of particle is found by deriving the velocity of that same particle with respect to  $t$ :

## Kinetic energy

Kinetic stands for motion, so the kinetic energy of a particle is the energy it gains through movement. Energy is a scalar quantity so we'll model it using a *FunExpr*.

```

type Energy = FunExpr

```

In classical mechanics the kinetic energy of particle is equal to  $\frac{1}{2}$  of the product of its mass and the square of its velocity. In mathematical notation this would be.

$$E_k = mv^2$$

But what does it mean to take the square of speed? We know of an operator to take the square of a number, but velocity is not a number it's a vector. So there seems to be a double meaning to what taking the square of something actually means depending on the type of the expression.

More concrete what we actually mean when we say the square of vector'' is to take the dot product of the vector with itself. This may seem fairly straightforward and obvious on paper but when we start working with types this overloading of the wordsquare'' looks a bit strange. For numbers it has the type

```

square :: Num a => a -> a

```

but for vectors it has the type

```

square :: Num a => Vector a -> a

```



same name, vastly different meanings. If we were to encode this double meaning of the function we would have to create a type class which would encode this ability to be squared and then make Num and Vector instances of this class.

But enough yammering about types and double meanings! Let's actually define the function for taking the square of a vector and move on with our lives.

```
square :: VectorE -> FunExpr
square v = dotProd v v
```

Pretty straightforward. So with this out of the way we can now define the function for calculating the kinetic energy of a particle, again this is almost the same as just writing down the mathematical formula with the added benefit of types.

```
kineticEnergy :: Particle -> Energy
kineticEnergy p = Const 0.5 * m * v2
  where
    m = mass p
    v = velocity p
    v2 = square v
```

## Potential energy

In classical mechanics potential energy is defined as the energy possessed by a particle because of its position relative to other particles, its electrical charge and other factors. This means that to actually calculate the potential energy of a particle we'd have to take into account all other particles that populate the system no matter how far apart they are.

## Potential energy near Earth

Thankfully if we're close to the Earth's gravitational field it's ok to simplify this problem and only take into account the Earth's gravitational pull since all other factors are negligible in comparison.

The potential energy for particles affected by gravity is defined with mathematical notation as:

$$E_p = m \cdot g \cdot h$$

where  $m$  is the mass of the particle,  $h$  its height, and  $g$  is the acceleration due to gravity ( $9.82 \text{ m/s}^2$ ).

```
potentialEnergy :: Particle -> Energy
```

```
potentialEnergy p = m * g * h
```

```
where
```

```
m          = mass p
```

```
g          = Const 9.82
```

```
(V3 _ _ h) = pos p
```

## Work

If a constant force  $\vec{F}$  is applied to a particle that moves from position  $\vec{r}_1$  to  $\vec{r}_2$  then the *work* done by the force is defined as the dot product of the force and the vector of displacement. In mathematical notation this is written as

$$W = \vec{F} \cdot \Delta\vec{r}$$

where  $\Delta\vec{r} = \vec{r}_2 - \vec{r}_1$ .

The work-energy theorem states that for a particle of constant mass  $m$ , the total work  $W$  done on the particle as it moves from position  $r_1$  to  $r_2$  is equal to the change in kinetic energy  $E_k$  of the particle:

$$W = \Delta E_K = E_{k,2} - E_{k,1} = \frac{1}{2}m(\vec{v}_2^2 - \vec{v}_1^2)$$

Let's codify this theorem:

PS: This used to work just fine, but it no longer does since the switch to FunExpr. Problem probably lies somewhere in SyntaxTree

```
prop_WorkEnergyTheorem :: Mass -> VectorE -> VectorE -> IO Bool
```

```
prop_WorkEnergyTheorem m v1 v2 = prettyEqual deltaEnergy (kineticEnergy
  displacedParticle)
```

```
where
```

```
particle1 = P v1 m -- | Two particles with the same mass
```

```
particle2 = P v2 m -- | But different position vector
```

```
-- |           E_k,2           - E_k,1
```

```
deltaEnergy = kineticEnergy particle2 - kineticEnergy particle1
displacedParticle = P (v2 - v1) m
```

```
-- Test values
```

```
v1 = V3 (3 :* Id) (2 :* Id) (1 :* Id)
v2 = V3 0 0 (5 :* Id)
v3 = V3 0 (3 :* Id) 0 :: VectorE
v4 = V3 2 2 2 :: VectorE
m = 5
p1 = P v1 m
p2 = P v2 m
dE = kineticEnergy p2 - kineticEnergy p1
p3 = P (v2 - v1) m
```

## Law of universal gravitation

Newton's law of universal gravitation states that a particle attracts every other particle in the universe with a force that is directly proportional to the product of their masses, and is inversely proportional to the square of the distance between their centers.

This means that every particle with mass attracts every other particle with mass by a force pointing along the line intersecting both points.

There is an equation for calculating the magnitude of this force which states with math what we stated in words above:

$$F = G \frac{m_1 m_2}{r^2}$$

Where  $F$  is the magnitude of the force,  $m_1$  and  $m_2$  are the masses of the objects interacting,  $r$  is the distance between the centers of the masses and  $G$  is the gravitational constant.

The gravitational constant has been finely approximated through experiments and we can state it in our code like this:

```
type Constant = FunExpr

gravConst :: Constant
gravConst = 6.674 * (10 ** (-11))
```

Now we can codify the law of universal gravitation using our definition of particles.

```
lawOfUniversalGravitation :: Particle -> Particle -> FunExpr
lawOfUniversalGravitation p1 p2 = gravConst * ((m_1 * m_2) / r2)
  where
    m_1 = mass p1
    m_2 = mass p2
    r2 = square $ pos p2 - pos p1
```

If a particles position is defined as a vector representing its displacement from some origin O, then its heigh should be x. Or maybe it should be the magnitude of the vector, if the gravitational force originates from O. Hmmm

This seems weird since I don't know what the frame of reference is...

```
potentialEnergy :: Particle -> Energy
potentialEnergy p = undefined
  where
    m          = mass p
    (V3 x _ _) = pos p
```

[src: [NewtonianMechanics/SingleParticle.lhs](#)] Previous: [Usage](#) [Table of contents](#) Next: [Teeter](#)

© Björn Werner, Erik Sjöström, Johan Johansson, Oskar Lundström (2018), GPL

# Learn You a **Physics** for Great Good!

>>> WORK IN PROGRESS <<<

## Examples / Teeter

[src:  
[Examples/Teeter.lhs](#)]

Previous: [Single particle mechanics](#)

[Table of contents](#)

Next: [Box on an incline](#)

```
module Examples.Teeter where
```

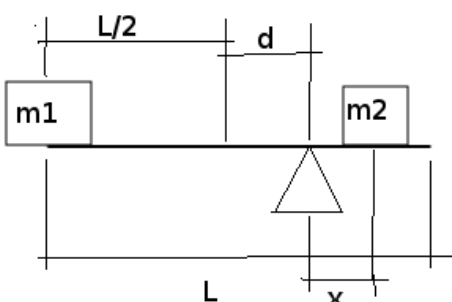
Exam excercise 3, 2017-01-13

```
import Dimensions.TypeLevel
import Dimensions.Quantity
import Prelude hiding (length)
```

Two boxes,  $m_1$  and  $m_2$ , rests on a beam in balance.

Known values:

```
beam_M = 1.0 # mass
m1 = 2.0 # mass
m2 = 5.0 # mass
d = 0.75 # length
beam_L = 5.0 # length
two = 2.0 # one
```



Teeter

Direct implication:

```
beam_left_L = (beam_L /# two) +# d
beam_right_L = beam_L -# beam_left_L
```

We want to be able to represent the torques.

A torque (sv. vridmoment) is defined as:

$$\tau = \text{distance from turning point} \cdot \text{force}$$

Since all force values will be composed of a mass and the gravitation, we can ignore the gravitation.

$$\tau = \text{distance from turning point} \cdot \text{mass}$$

$$m1\_torq = m1 \cdot \text{beam\_left\_L}$$

To get the beams torque on one side, we need to divide by 2 because the beam's torque is spread out linearly (the density of the beam is equal everywhere), which means the left parts mass centrum is of the left parts total length.

$$\text{beam}L_\tau = \text{beam}L_M \cdot \frac{\text{distance}}{2}$$

$$\text{beam}L_\tau = \frac{\text{beam left length}}{\text{beam length}} \cdot \text{beam}m_M \cdot \frac{\text{beam left length}}{2}$$

$$\text{beamL\_torq} = ((\text{beam\_left\_L} / \text{beam\_L}) \cdot \text{beam\_M}) \cdot (\text{beam\_left\_L} / 2)$$

$$\text{beamR\_torq} = ((\text{beam\_right\_L} / \text{beam\_L}) \cdot \text{beam\_M}) \cdot (\text{beam\_right\_L} / 2)$$

We make an expression for  $m2_\tau$ , which involves our unknown distance  $x$ .

$$m2_\tau = m2 \cdot x$$

For the teeter to be in balance, both sides torques should be equal.

$$\text{Left side torque} = \text{Right side angular torque}$$

We try to break out  $m2_\tau$  and then  $x$ .

$$m1_\tau + \text{beam}L_\tau = m2_\tau + \text{beam}R_\tau$$

$$m1_\tau + \text{beam}L_\tau - \text{beam}R_\tau = m2_\tau$$

$$\frac{m1_\tau + \text{beam}L_\tau - \text{beam}R_\tau}{m2} = x$$

Our solution:

```
x = (m1_torq +# beamL_torq -# beamR_torq) /# m2
```

Security check:

```
m2_torq = m2 *# x
```

```
left_side_torque = m1_torq +# beamL_torq  
right_side_torque = m2_torq +# beamR_torq
```

[src:  
[Examples/Teeter.lhs](#)]

Previous: [Single particle  
mechanics](#)

[Table of  
contents](#)

Next: [Box on an  
incline](#)

© Björn Werner, Erik Sjöström, Johan Johansson, Oskar Lundström (2018), GPL

# Learn You a **Physics** for Great Good!

>>> WORK IN PROGRESS <<<

## Examples / Box on an incline

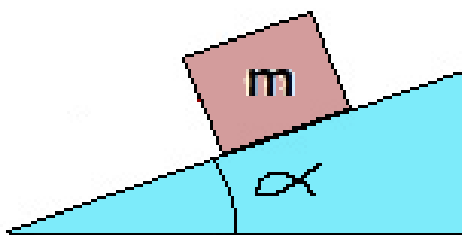
[src: [Examples/Box\\_incline.lhs](#)] Previous: [Teeter](#) [Table of contents](#) Next: [Table of contents](#)

Improvement: notation formulas tests

## Box on an incline

```
import Vector.Vector
```

All vectors are in newton.



Incline

Notation:  $fg$  = gravitational acceleration

$m$  = mass of box

```
fg = V2 0 (-10)
m = 2
```

```
unit_normal :: Double -> Vector2 Double
unit_normal a = V2 (cos a) (sin a)
```

Force against the incline from the box:

```
f_l_ :: Vector2 Double -> Angle -> Vector2 Double
f_l_ fa a = scale ((magnitude fa) * (cos a)) (unit_normal (a-(pi/2)))
```



The normal against the incline:

```
fn :: Vector2 Double -> Angle -> Vector2 Double
fn fa a = negate (f_l_ fa a)
```

Friction free incline:

Resulting force:

```
fr :: Vector2 Double -> Angle -> Vector2 Double
fr fa a = (fn fa a) + fa
```

With friction:

$$F_{friction} = \mu * F_{normal} \iff \mu = \frac{F_{friction}}{F_{normal}}$$

```
us = 0.5
```

```
uk = 0.4
```

Add image how friction depends if there is movement.

Friction

Friction

```
type FricConst = Double
```

Friction:

friks = Fn \* us, us = friction static

frikk = Fn \* uk, uk = friction kinetic

We have the normal force and only needs the constants.

The current speed does not affect the friction.

```
motsalar :: FricConst -> Vector2 Double -> Scalar
motsalar u f = u * (magnitude f)
```

Från en rörelse eller vekt, fixa komplementet

```
enh_vekt :: Vector2 Double -> Vector2 Double
enh_vekt v | magnitude v == 0 = (V2 0 0)
           | otherwise = scale (1 / (magnitude v)) v
```

```
motkrafts :: FricConst -> Scalar -> Vector2 Double -> Vector2 Double
motkrafts u s v = scale (u * s) (negate (enh_vekt v))
```

```
motkraftv :: FricConst -> Vector2 Double -> Vector2 Double -> Vector2 Double
motkraftv u n v = scale (u * (magnitude n)) (negate (enh_vekt v))
```

Now we just need to sum the force vectors:

```
fru :: Vector2 Double -> Angle -> FricConst -> Vector2 Double
fru fa a u = (fr fa a) + (motkraftv u (fn fa a) (fr fa a))
```

```
fru' :: Vector2 Double -> Angle -> FricConst -> Vector2 Double
fru' fa a u = (motkraftv u (fn fa a) (fr fa a))
```

[src: [Examples/Box\\_incline.lhs](#)] Previous: [Teeter](#) [Table of contents](#) Next: [Table of contents](#)

© Björn Werner, Erik Sjöström, Johan Johansson, Oskar Lundström (2018), GPL

# Learn You a **Physics** for Great Good!

>>> **WORK IN PROGRESS** <<<

## Examples / Teeter

[src: [Examples/Teeter.lhs](#)] Previous: [Single particle mechanics](#) [Table of contents](#) Next: [Box on an incline](#)

```
module Examples.Teeter where
```

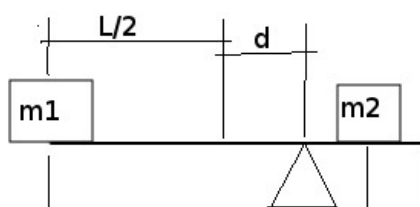
Exam exercise 3, 2017-01-13

```
import Dimensions.TypeLevel
import Dimensions.Quantity
import Prelude hiding (length)
```

Two boxes,  $m_1$  and  $m_2$ , rests on a beam in balance.

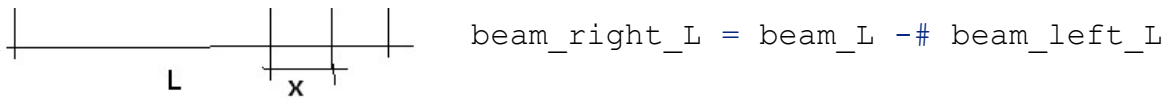
Known values:

```
beam_M = 1.0 # mass
m1 = 2.0 # mass
m2 = 5.0 # mass
d = 0.75 # length
beam_L = 5.0 # length
two = 2.0 # one
g = 9.0 # acceleration
```



Direct implication:

```
beam_left_L = (beam_L /# two) +# d
```



We want to be able to represent the torques.

A torque (sv. vridmoment) is defined as:

$$\tau = \text{distance from turning point} \cdot \text{force}$$

(soon not to be) Since all force values will be composed of a mass and the gravitation, we can ignore the gravitation.

$$\tau = \text{distance from turning point} \cdot \text{mass} \cdot \text{gravitation}$$

$$m1\_torq = m1 \cdot (g \cdot \text{beam\_left\_L})$$

To get the beams torque on one side, we need to divide by 2 because the beam's torque is spread out linearly (the density of the beam is equal everywhere), which means the left parts mass centre is *half the distance* of the left parts total length.

$$\text{beam}L_{\tau} = \text{beam}L_M \cdot \text{gravity} \cdot \frac{\text{beam left length}}{2}$$

where

$$\text{beam}L_M = \frac{\text{beam left length}}{\text{beam length}} \cdot \text{beam}_M$$

$$\text{beam}L\_torq = ((\text{beam\_left\_L} / \# \text{ beam\_L}) \cdot (\text{beam\_M} \cdot g)) \cdot (\text{beam\_left\_L} / \# \text{ two})$$

$$\text{beam}R\_torq = ((\text{beam\_right\_L} / \# \text{ beam\_L}) \cdot (\text{beam\_M} \cdot g)) \cdot (\text{beam\_right\_L} / \# \text{ two})$$

We make an expression for  $m2_{\tau}$ , which involves our unknown distance  $x$ .

$$m2_{\tau} = m2 \cdot x$$

For the teeter to be in balance, both sides torques should be equal.

$$\text{Left side torque} = \text{Right side angular torque}$$

We try to break out  $m2_{\tau}$  and then  $x$ .

$$m1_{\tau} + beamL_{\tau} = m2_{\tau} + beamR_{\tau}$$

$$m1_{\tau} + beamL_{\tau} - beamR_{\tau} = m2_{\tau}$$

$$\text{the distance } x = \frac{m2_{\tau}}{m2 \cdot gravitation}$$

Our solution:

$$x = (m1\_torq +\# beamL\_torq -\# beamR\_torq) /\# (m2 \*\# g)$$

Security check:

$$m2\_torq = (m2 \*\# g) \*\# x$$

$$\text{left\_side\_torque} = m1\_torq +\# beamL\_torq$$

$$\text{right\_side\_torque} = m2\_torq +\# beamR\_torq$$

We can control that both sides total torque are equal, and that the dimensions of  $x$  is a length.

[src: [Examples/Teeter.lhs](#)] Previous: [Single particle mechanics](#) [Table of contents](#) Next: [Box on an incline](#)

© Björn Werner, Erik Sjöström, Johan Johansson, Oskar Lundström (2018), GPL

# Learn You a **Physics** for Great Good!

>>> **WORK IN PROGRESS** <<<

## Examples / Box on an incline

[src: [Examples/Box\\_incline.lhs](#)]

Previous: [Teeter](#)

[Table of contents](#)

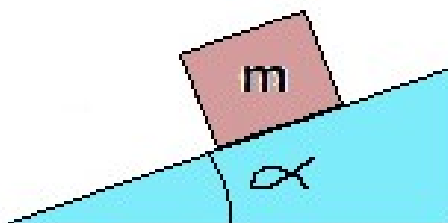
Next: [Table of contents](#)

## Box on an incline

```
import Vector.Vector
import Text.Printf
```

A box with the mass 2kg is resting on an incline. The task is to determinate the resulting force given an angle of the incline.

When calculating with forces in this example we will use force vectors. This way we can add and subtract different forces together. We can also scale a force with a scalar.



$\dot{g}$  = *gravitational acceleration*

$m$  = *mass of box*

```
g :: Vector2 Double
g = V2 0 (-10)
```

```
m :: Double
m = 2
```

We scale the gravitational acceleration vector with the mass of the box to get a vector representing

the gravitational force.

$$\dot{F}_g = m \cdot \dot{g}$$

```
fg :: Vector2 Double
fg = scale m g

alpha :: Angle
alpha = pi/4
```

A unit vector that we get from a radian  $a$ .

```
angle_to_unit_vec :: Angle -> Vector2 Double
angle_to_unit_vec angle = V2 (cos angle) (sin angle)
```

Force against the incline from the box we denote as  $\dot{F}_\perp$ . Since the force is perpendicular against the incline of the surface, we need to create the unit vector for  $\dot{F}_\perp$  by adding  $-(\pi/2)$  radians.

This vector we then scale up, with the magnitude of the gravitational force multiplied by  $\cos(\text{angle})$ , which gives how much the gravitational force that affects  $\dot{F}_\perp$ .

There are two edge cases. If the incline is flat, it means the incline has the angle 0, and  $\cos(0) = 1$ , which means  $\dot{F}_g = \dot{F}_\perp$ . If the incline is fully tilted however  $\frac{\pi}{2}$ , it means that  $\cos(\frac{\pi}{2}) = 0$  and the gravitational force on the box does not result in any force against the incline.

```
f_perp :: Vector2 Double -> Angle -> Vector2 Double
f_perp fa angle = scale ((magnitude fa) * (cos angle))
  (angle_to_unit_vec (angle-(pi/2)))
```

The normal force  $\dot{F}_n = -\dot{F}_\perp$  supporting the box from the incline:

```
fn :: Vector2 Double -> Angle -> Vector2 Double
fn fa angle = negate (f_perp fa angle)
```

The *resulting force*  $\dot{F}_r$  is then the normal force from the incline plus the gravitational force.

$$\dot{F}_r = \dot{F}_n + \dot{F}_g$$

```
fr :: Vector2 Double -> Angle -> Vector2 Double
fr fa angle = (fn fa angle) + fa
```

We have now determined a way to get the resulting force from a given angle. Below we show how we can use it.

```
*Main> fr fg (pi/4)
(-10.000000000000002, -10.0)
```

With the use of a rounding printer (defined last in this example) we can round the output.

```
*Main> print_vec_2dec $ fr fg (pi/4)
(-10.00,-10.00)
```

Here we can see, that  $\vec{F}_r$  on the box is pointing down-left in the sense of a coordinate system. To know it's magnitude, we can do:

```
magnitude $ fr fg (pi/4)
14.142135623730953
```

Or with rounding:

```
print_2dec $ magnitude $ fr fg (pi/4)
14.14
```

## Now we try to tackle the problem with friction:

$$F_{friction} = \mu * F_{normal} \iff \mu = \frac{F_{friction}}{F_{normal}}$$

There are two different kinds of friction. One where the object is standing still on the surface, and the other where the object is sliding on the surface. In the case where the object is standing still, it remains still until the force applied on the object is greater than the maximum possible friction between the object and the surface. Once that level is surpassed, the object starts to slide.



When the object is sliding the maximum friction between the surface and the object is slightly less, as illustrated in the figure below.



$$F_{static\ friction} = \mu_{static} \cdot F_{normal}$$

$$F_{kinetic\ friction} = \mu_{kinetic} \cdot F_{normal}$$

Given that we know if the box is already moving or not, we can choose the corresponding friction constant and calculate the friction force between the box and the incline. The direction of the friction force will then be in the opposite direction of the resulting force we determined in the previous section.

We will use a type definition to clarify parameters.

```
type FricConst = Double
```

We will also need to use the unit vector of the resulting force from the previous section.

```
unit_vec :: Vector2 Double -> Vector2 Double
unit_vec v | magnitude v == 0 = (V2 0 0)
           | otherwise = scale (1 / (magnitude v)) v
```

## For a box initially at rest:

We let this function compute the friction. If the magnitude of the friction is greater than the magnitude of the resulting force without friction, it means the box will stand still or that the magnitudes can be treated as equally big.

```
ff :: Vector2 Double -> Scalar -> Scalar -> FricConst -> Vector2
   Double
ff fr magn_l_ magn_fr u = scale magn_fric (negate (unit_vec fr))
   where
       magn_fric | magn_l_ * u > magn_fr = magn_fr
                 | otherwise = magn_l_ * u
```

Now we just need to add the friction vector to our old  $\vec{F}_r$ . We call this function `fru`, from the

constant  $\mu$ :

```
fru :: Vector2 Double -> Angle -> FricConst -> Vector2 Double
fru fa a u = (fr fa a) + (ff (fr fa a) magn_l_ magn_fr u)
  where
    magn_l_ = magnitude (f_l_ fa a)
    magn_fr = magnitude (fr fa a)
```

Testing for  $u = 0$ . In this case both “fr” and “fru” should give the same resulting vector:

```
*Main> print_vec_2dec $ fr fg (pi/3)
(-8.66,-15.00)

*Main> print_vec_2dec $ fru fg (pi/3) 0
(-8.66,-15.00)
```

They should also give the same if the incline is flat.

```
*Main> print_vec_2dec $ fr fg (0)
(-0.00,0.00)

*Main> print_vec_2dec $ fru fg (0) 1
(-0.00,0.00)
```

Lastly one should see the edge case where the angle is  $\frac{\pi}{4}$ . The magnitude of the normal and the resulting force from the previous section becomes the same (try and control it!). This also means that if  $\mu_{static} = 1$  the magnitude of the friction force becomes *equal* to the magnitude of the resulting force without friction. The consequence of this will be that the friction force is just enough to prevent the box from sliding. If we increase the angle just a little, the box starts sliding.

```
*Main> print_vec_2dec $ fru fg (pi/4) 1
(-0.00,0.00)

*Main> print_vec_2dec $ fru fg (0.01 + pi/4) 1
(-0.20,-0.20)
```

In this case it may be beneficial to see more decimals, given that it should slide more downwards (down on the y-axis) than to the left (left on the x-axis).

```
fru fg (0.01 + pi/4) 1
(-0.197986733599107, -0.201986600267551)
```

## Friction for a box in motion:

This one is for you to solve. A box in motion is always affected by the friction (except for one particular edge case).

## Pretty printing:

```
print_2dec :: Double -> IO ()
print_2dec a = putStrLn $ printf "%.2f" a

print_vec_2dec :: Vector2 Double -> IO ()
print_vec_2dec (V2 a b) = putStrLn $ "(" ++ (printf "%.2f" a) ++ ", "
  ++ (printf "%.2f" b) ++ ")"
```

[src: [Examples/Box\\_incline.lhs](#)]    Previous: [Teeter](#)    [Table of contents](#)    Next: [Table of contents](#)

© Björn Werner, Erik Sjöström, Johan Johansson, Oskar Lundström (2018), GPL