Fortgeschrittene Funktionale Programmierung in Haskell

Universität Bielefeld, Sommersemester 2015

Jonas Betzendahl & Stefan Dresselhaus

Orga-Krams

Heute ist die letzte Vorlesung, die Deadline für die Projekte ist am Freitag, den 18.09.2015. Ab sofort nehmen wir Abgaben für die Projekte entgegen.
Für eine Abgabe schreibt bitte eine Mail an beide Tutoren (`sdressel@techfak...` und `jbetzend@techfak...`) mit entweder dem kompletten Quellcode oder einem Link auf einen Tag auf `github.com`.

Orga-Krams

Heute ist die letzte Vorlesung, die Deadline für die Projekte ist am Freitag, den 18.09.2015. Ab sofort nehmen wir Abgaben für die Projekte entgegen.
Für eine Abgabe schreibt bitte eine Mail an beide Tutoren (`sdressel@techfak...` und `jbetzend@techfak...`) mit entweder dem kompletten Quellcode oder einem Link auf einen Tag auf `github.com`.

Besonderer Dank geht an Alexander Sczyrba und Mario Botsch, dafür dass es uns möglich gemacht wurde, diese Vorlesung überhaupt zu halten, an alle, die uns bei der Durchführung geholfen haben und natürlich an alle, die mitgemacht haben.

Übersicht I

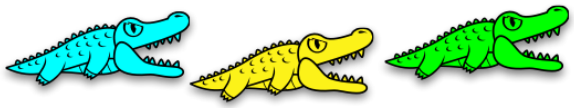
Alligator Eggs

Idee & Bilder: Bret Victor
http://worrydream.com/AlligatorEggs/

Wir betrachten heute ein Spiel, das gleichzeitig bunt und putzig ist und uns erlaubt, etwas interessantes zu lernen! Es gibt...

Wir betrachten heute ein Spiel, das gleichzeitig bunt und putzig ist und uns erlaubt, etwas interessantes zu lernen! Es gibt...

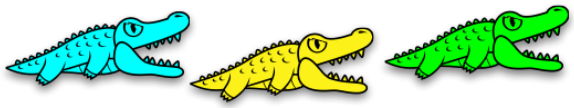
- **Hungrige Alligatoren**



Hungrige Alligatoren sind hungrig! Sie fressen alles, was ihnen vor's Maul kommt. Sie bewachen aber außerdem ihre Familien.

Wir betrachten heute ein Spiel, das gleichzeitig bunt und putzig ist und uns erlaubt, etwas interessantes zu lernen! Es gibt...

- **Hungrige Alligatoren**

  Hungrige Alligatoren sind hungrig! Sie fressen alles, was ihnen vor's Maul kommt. Sie bewachen aber außerdem ihre Familien.

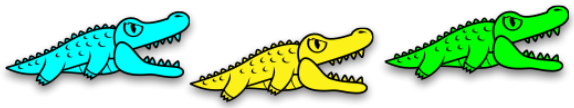
- **Alte Alligatoren**

  Diese Alligatoren haben genug gegessen und bewachen nur noch ihre Familien.

Wir betrachten heute ein Spiel, das gleichzeitig bunt und putzig ist und uns erlaubt, etwas interessantes zu lernen! Es gibt...

- **Hungrige Alligatoren**

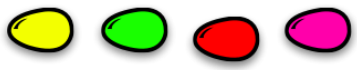
  Hungrige Alligatoren sind hungrig! Sie fressen alles, was ihnen vor's Maul kommt. Sie bewachen aber außerdem ihre Familien.

- **Alte Alligatoren**

  Diese Alligatoren haben genug gegessen und bewachen nur noch ihre Familien.

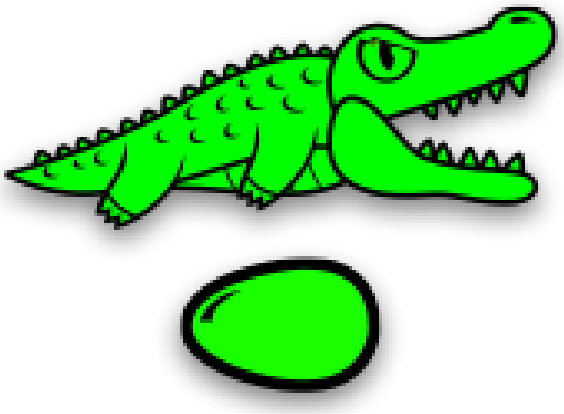
- **Alligatoreier**

  Aus Eiern schlüpfen demnächst neue Alligatorfamilien.

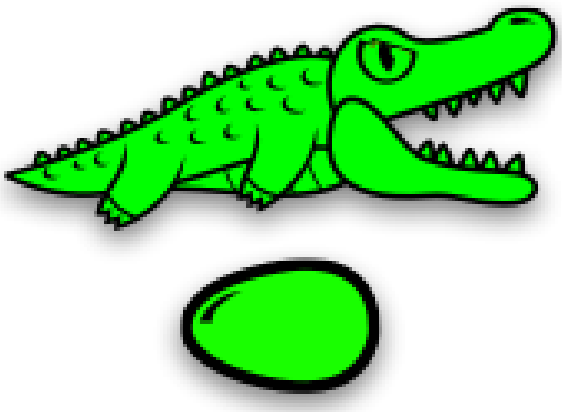
Familien

Alligatoren kommen in Familien daher. Hier ist eine:



Familien

Alligatoren kommen in Familien daher. Hier ist eine:



Hier ist noch nicht viel zu sehen, was wirklich interessiert.

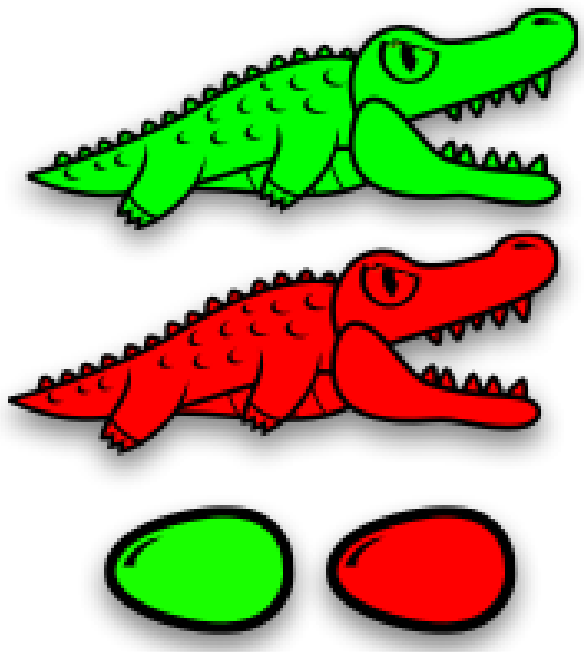
Nur ein grüner Alligator, der sein grünes Ei bewacht.
Ist er nicht süß?

Familien

Hier ist eine weitere Familie, dieses Mal mit mehr Mitgliedern.

Familien

Hier ist eine weitere Familie, dieses Mal mit mehr Mitgliedern.



Ein grüner und ein roter Alligator bewachen ein grünes und ein rotes Ei.

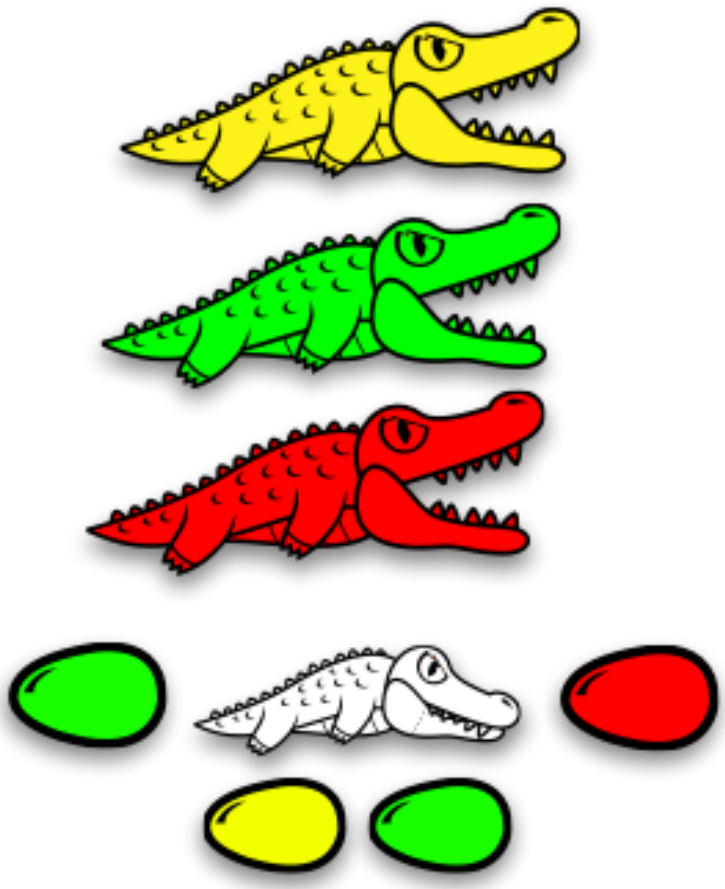
Oder anders formuliert:
Ein grüner Alligator bewacht einen roten Alligator und der rote Alligator bewacht die zwei Eier.

Familien

Dieses Mal haben wir eine richtige Großfamilie:

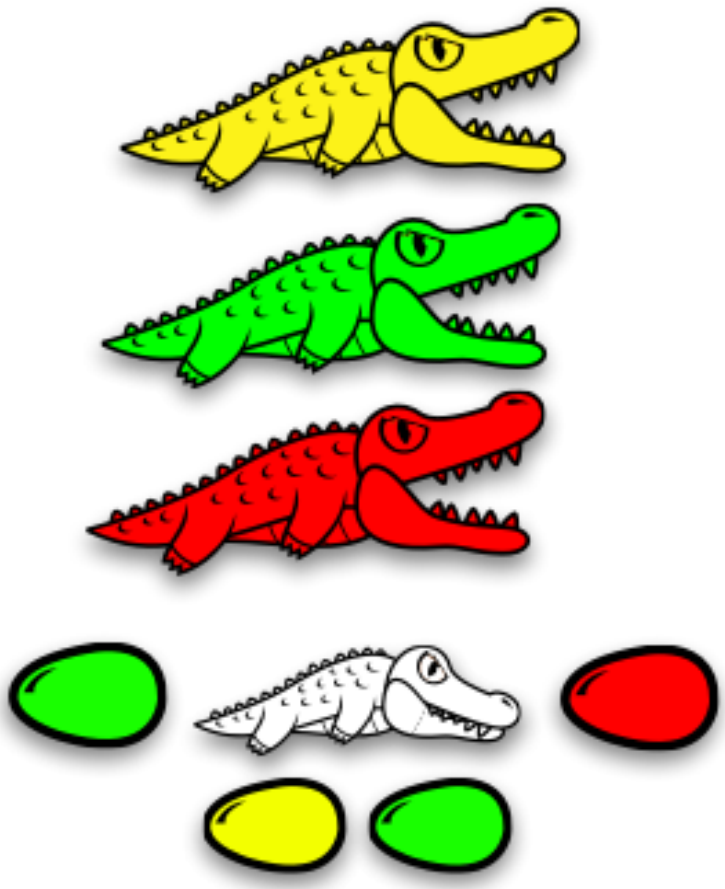
Familien

Dieses Mal haben wir eine richtige Großfamilie:



Familien

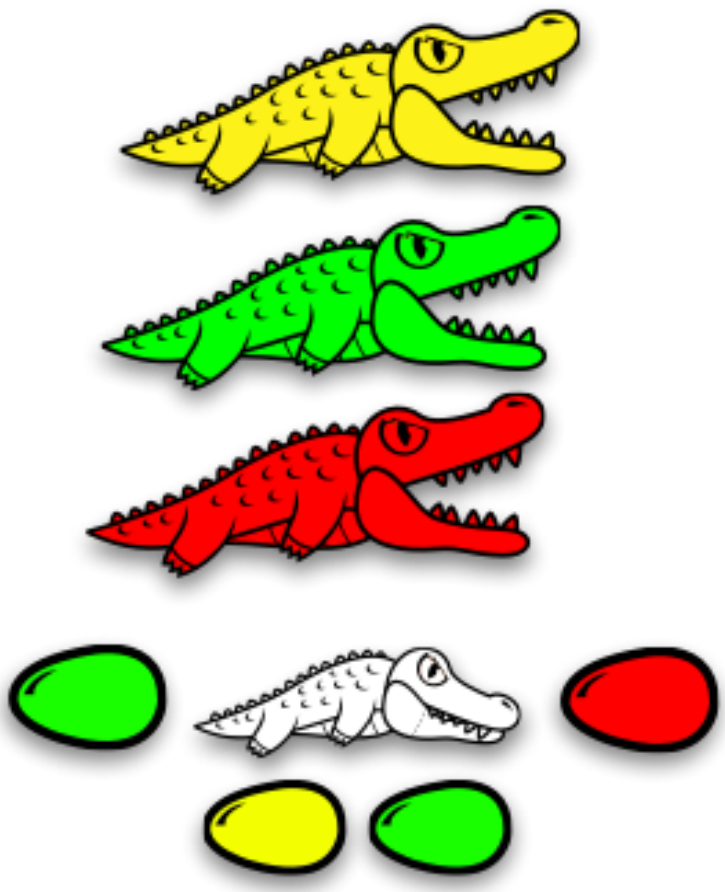
Dieses Mal haben wir eine richtige Großfamilie:



Hier haben wir drei hungrige Alligatoren, die Wache halten. Einen gelben, einen grünen und einen roten.

Familien

Dieses Mal haben wir eine richtige Großfamilie:

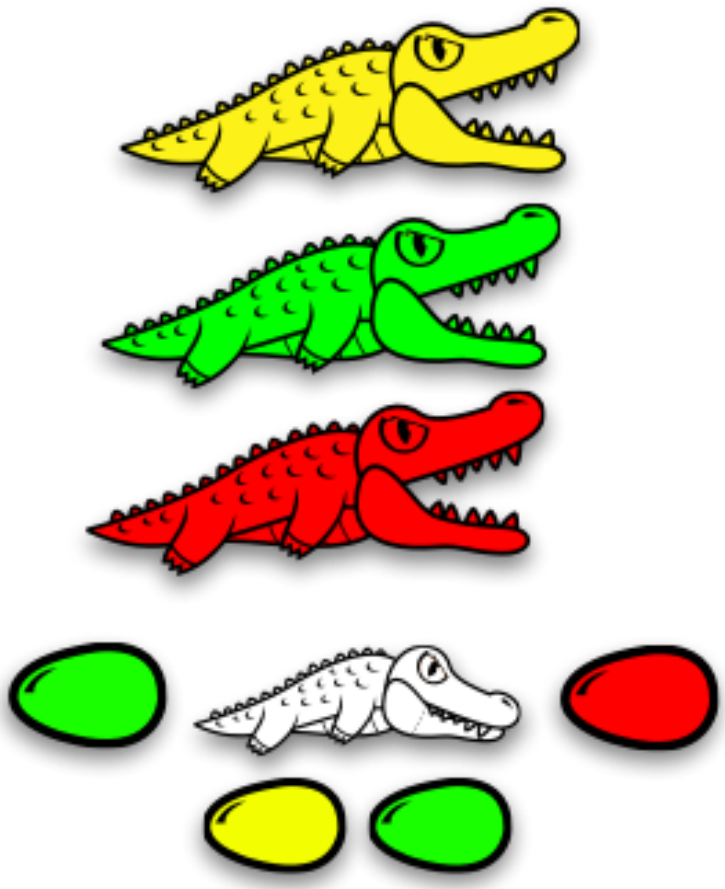


Hier haben wir drei hungrige Alligatoren, die Wache halten. Einen gelben, einen grünen und einen roten.

Sie bewachen drei Dinge: Ein grünes Ei, einen alten Alligator und ein rotes Ei.

Familien

Dieses Mal haben wir eine richtige Großfamilie:

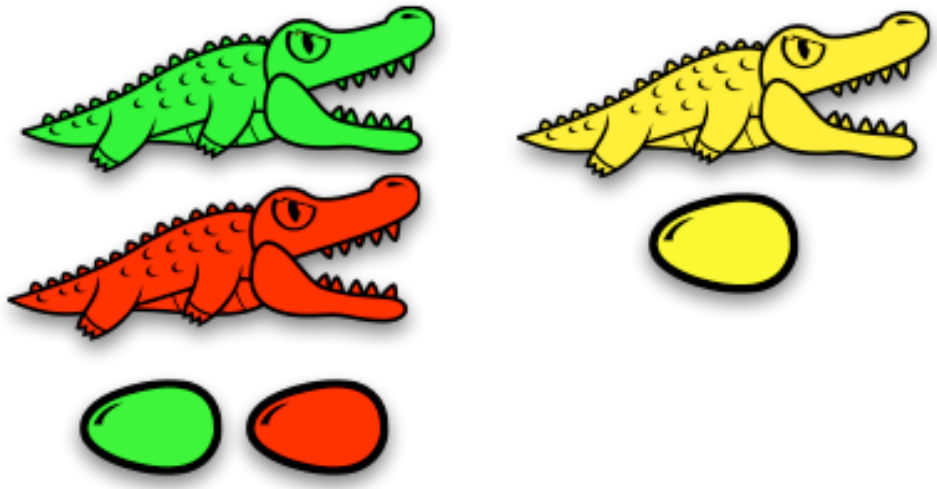


Hier haben wir drei hungrige Alligatoren, die Wache halten. Einen gelben, einen grünen und einen roten.

Sie bewachen drei Dinge: Ein grünes Ei, einen alten Alligator und ein rotes Ei.

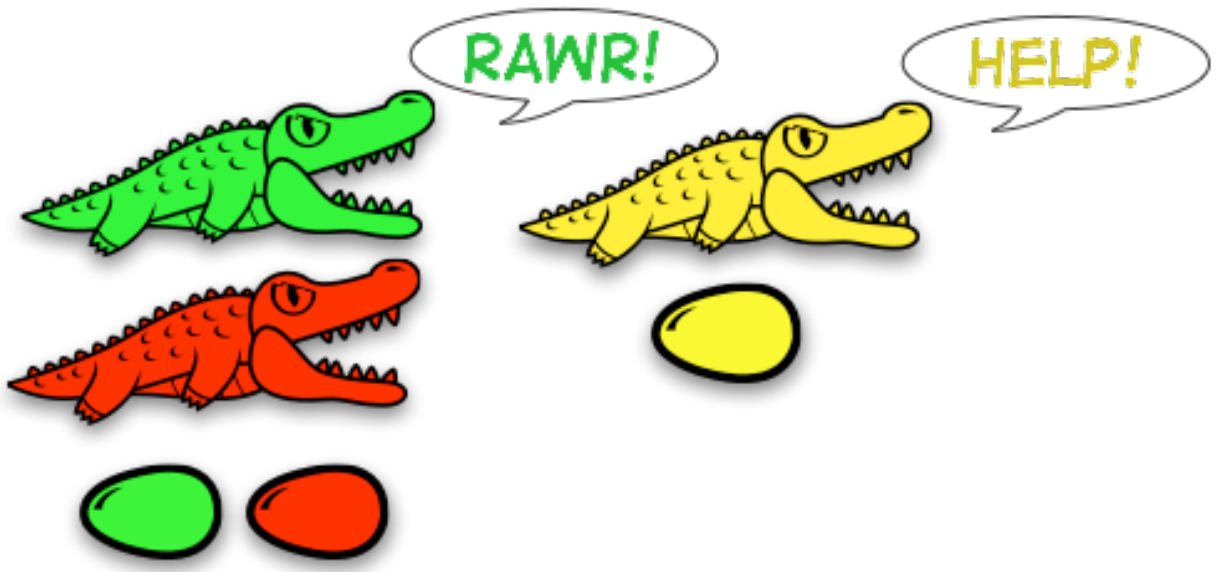
Der alte Alligator hingegen bewacht ein gelbes und ein grünes Ei.

Fressen und gefressen werden



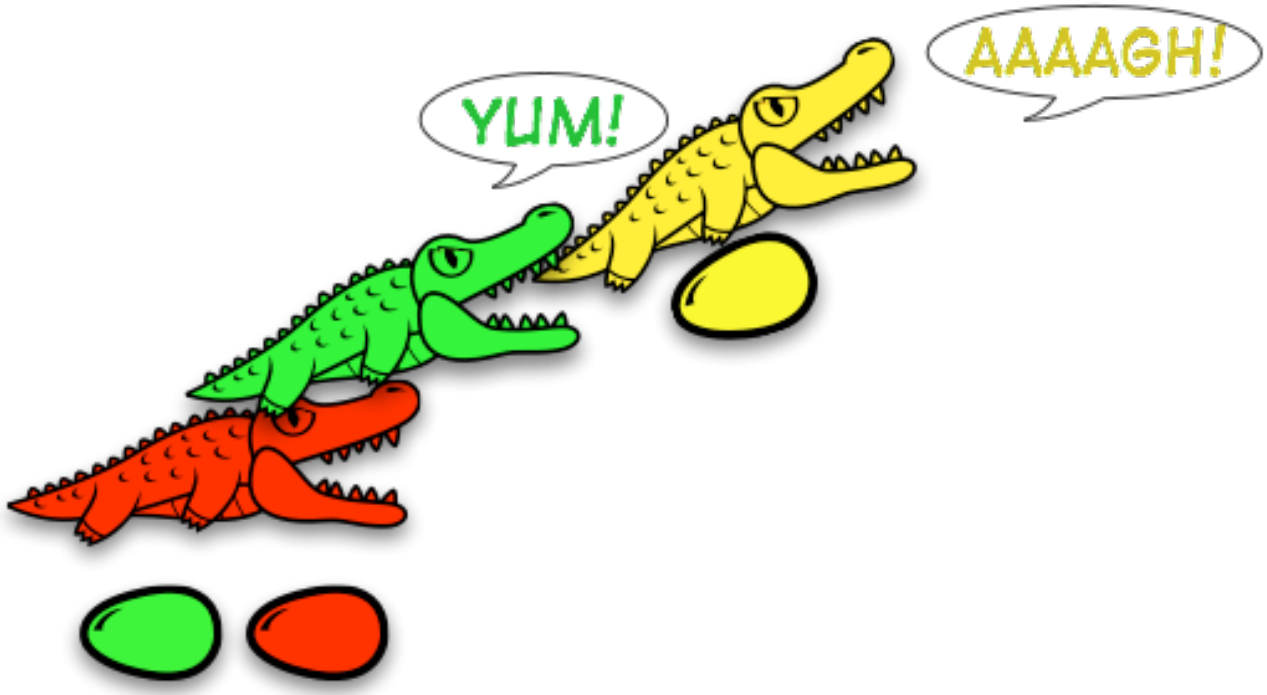
Hier wird es etwas ungemütlicher. Wir sehen hier zwei Familien nebeneinander.

Fressen und gefressen werden



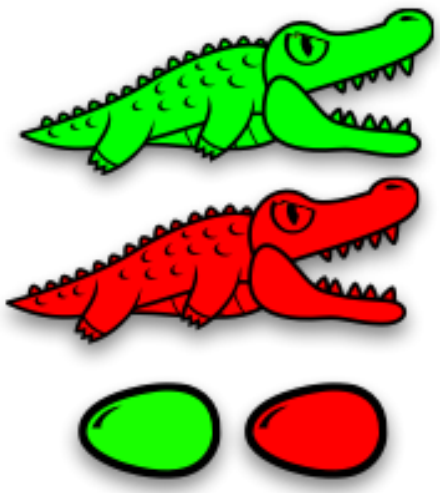
Hier wird es etwas ungemütlicher. Wir sehen hier zwei Familien nebeneinander. Der grüne Alligator ist *sehr* hungrig. . .

Fressen und gefressen werden



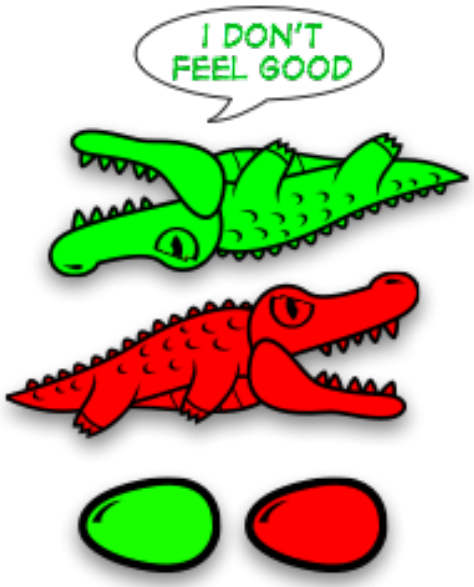
Hier wird es etwas ungemütlicher. Wir sehen hier zwei Familien nebeneinander. Der grüne Alligator ist *sehr* hungrig. . .

Fressen und gefressen werden



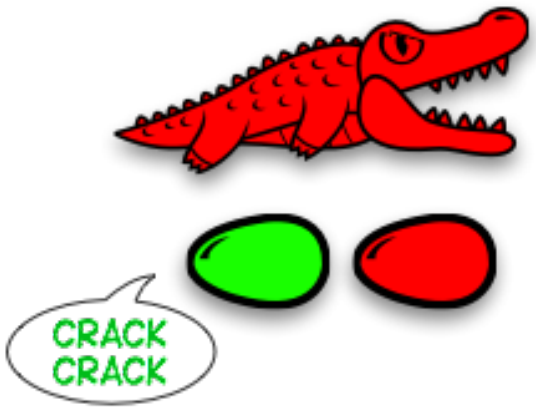
Der grüne Alligator hat die komplette gelbe Familie gefressen.

Fressen und gefressen werden



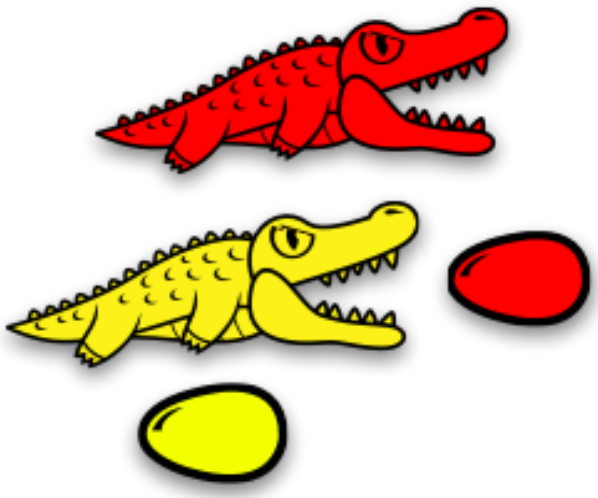
Der grüne Alligator hat die komplette gelbe Familie gefressen. Das war allerdings zu viel für seinen Magen. Er hat sich überfressen und stirbt.

Fressen und gefressen werden



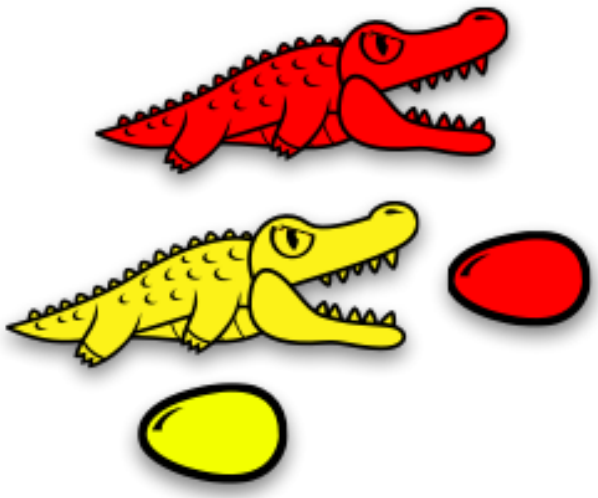
Was übrig bleibt ist der rote Alligator. Jetzt wo sein grüner Freund gestorben ist, fängt jedoch das grüne Ei an, zu schlüpfen.

Fressen und gefressen werden



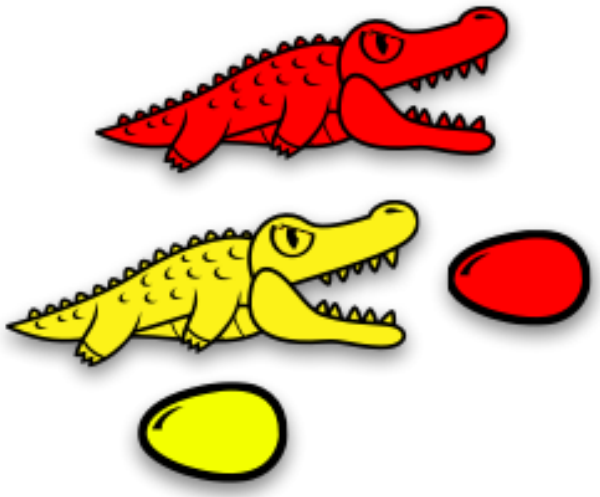
Was übrig bleibt ist der rote Alligator. Jetzt wo sein grüner Freund gestorben ist, fängt jedoch das grüne Ei an, zu schlüpfen.
Es schlüpft *exakt*, was der grüne Alligator gerade gegessen hat.
Das Wunder des Lebens!

Fressen und gefressen werden



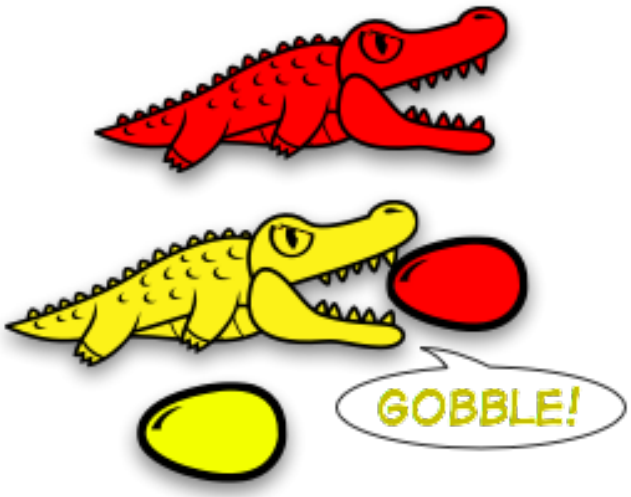
Was übrig bleibt ist der rote Alligator. Jetzt wo sein grüner Freund gestorben ist, fängt jedoch das grüne Ei an, zu schlüpfen.
Es schlüpft *exakt*, was der grüne Alligator gerade gegessen hat.
Das Wunder des Lebens!

Fressen und gefressen werden



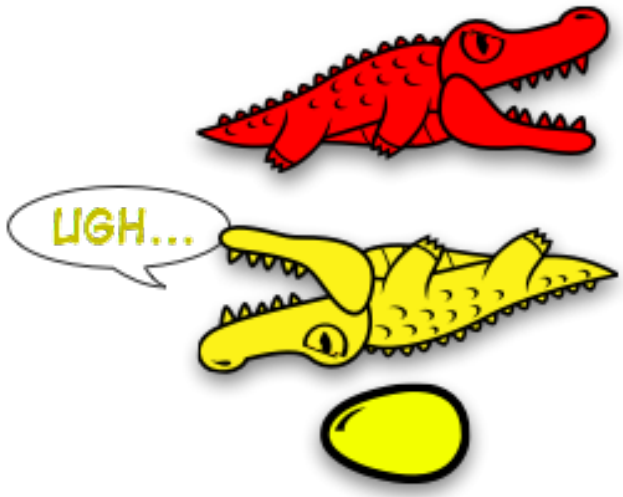
Jetzt ist also eine neue gelbe Familie geschlüpft.

Fressen und gefressen werden



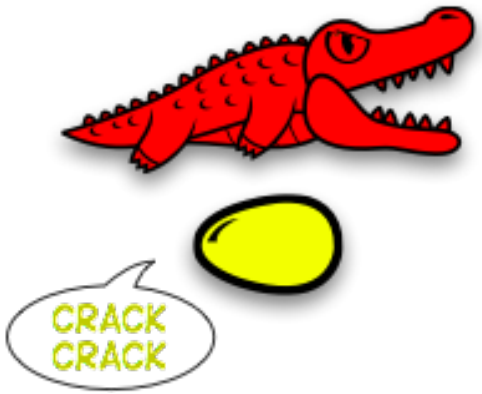
Jetzt ist also eine neue gelbe Familie geschlüpft. Allerdings ist dieser gelbe Alligator auch ziemlich hungrig...

Fressen und gefressen werden



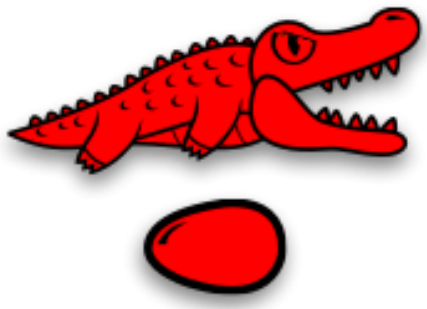
Jetzt ist also eine neue gelbe Familie geschlüpft. Allerdings ist dieser gelbe Alligator auch ziemlich hungrig...
Nachdem er das rote Ei gefressen hat, ist allerdings auch sein Magen schon zu voll und ihn ereilt das gleiche Schicksal wie den grünen Alligator.

Fressen und gefressen werden



Jetzt ist also eine neue gelbe Familie geschlüpft. Allerdings ist dieser gelbe Alligator auch ziemlich hungrig...
Nachdem er das rote Ei gefressen hat, ist allerdings auch sein Magen schon zu voll und ihn ereilt das gleiche Schicksal wie den grünen Alligator. Und auch aus diesem Ei schlüpft, was gerade gegessen wurde.

Fressen und gefressen werden



Jetzt ist also eine neue gelbe Familie geschlüpft. Allerdings ist dieser gelbe Alligator auch ziemlich hungrig...
Nachdem er das rote Ei gefressen hat, ist allerdings auch sein Magen schon zu voll und ihn ereilt das gleiche Schicksal wie den grünen Alligator. Und auch aus diesem Ei schlüpft, was gerade gegessen wurde.
Hier endet das Drama, da es nichts mehr zum Fressen gibt.

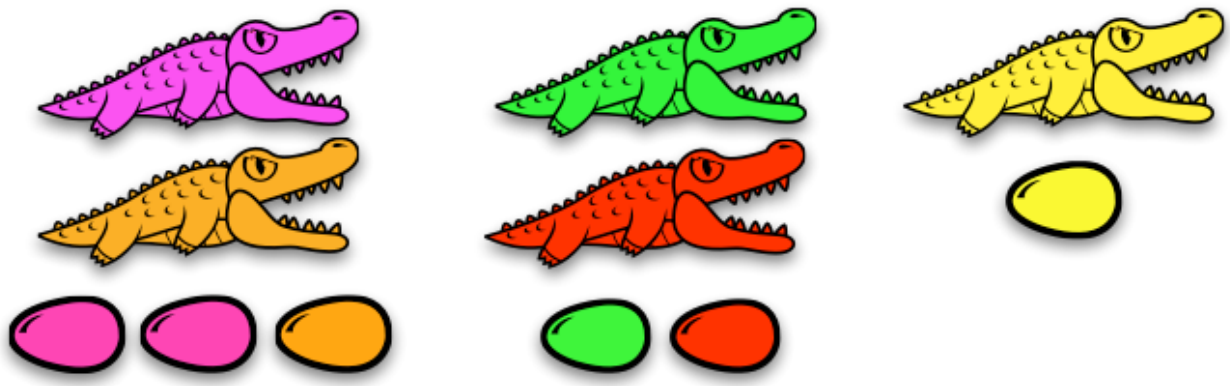
Die Essensregel

Wir können jetzt eine erste „formale“ Regel für dieses System aufstellen:

Die Essensregel

Wir können jetzt eine erste „formale“ Regel für dieses System aufstellen:

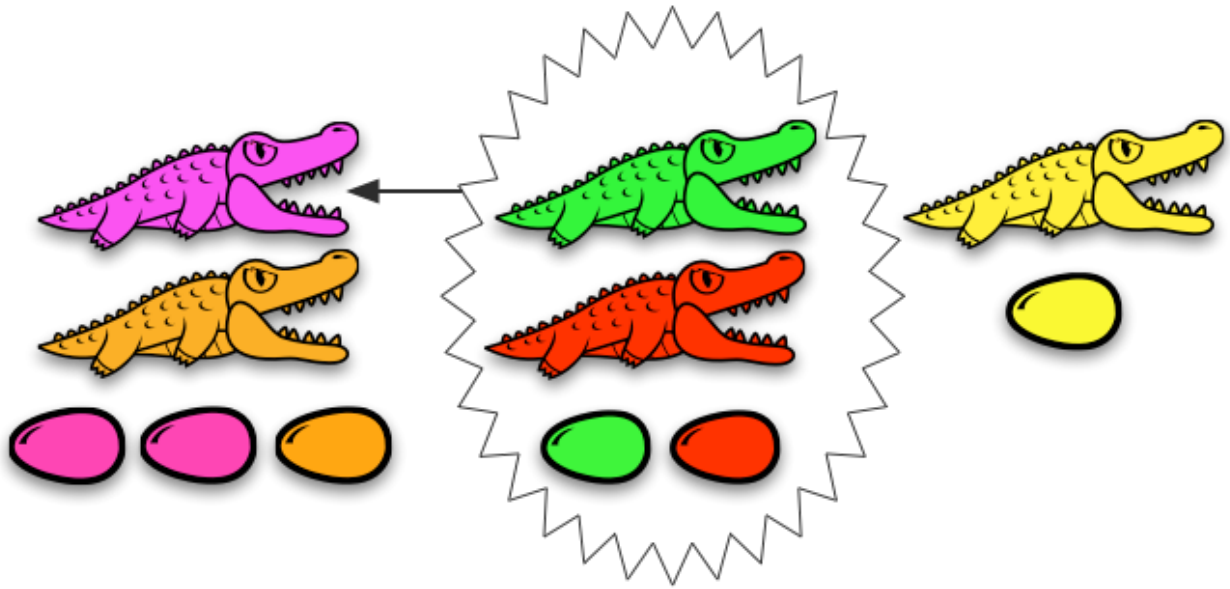
Wenn wir Alligatorfamilien nebeneinander haben, . . .



Die Essensregel

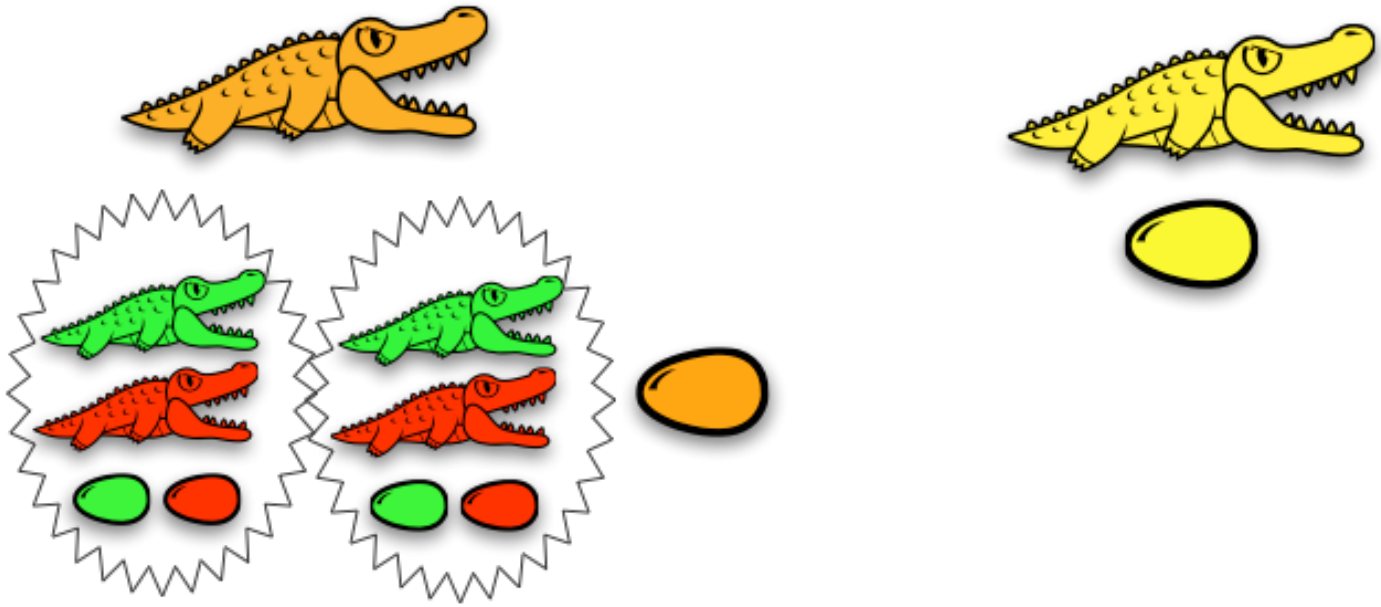
Wir können jetzt eine erste „formale“ Regel für dieses System aufstellen:

Wenn wir Alligatorfamilien nebeneinander haben, frisst der Alligator links oben die Familie rechts neben ihm.

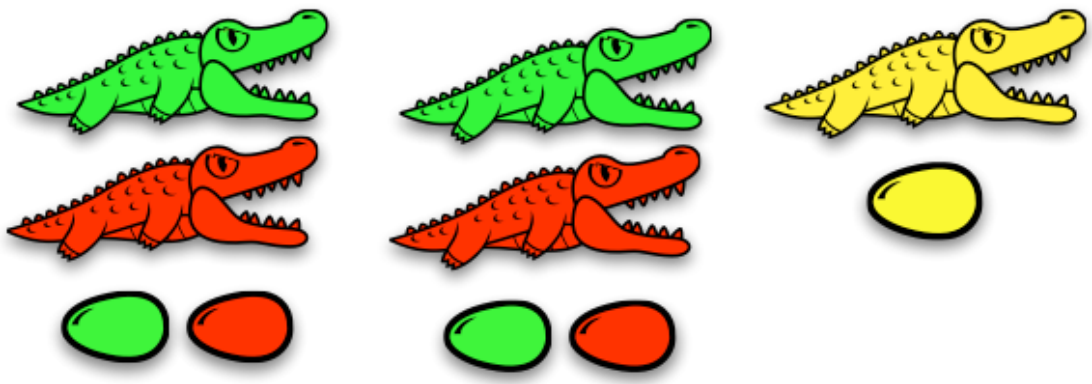


Die Essensregel

Wenn wir Alligatorfamilien nebeneinander haben, frisst der Alligator links oben die Familie rechts neben ihm. Dieser Alligator stirbt. Bewacht seine Familie jedoch Eier seiner Farbe, schlüpft aus *jedem* dieser Eier, was er gerade noch verspeist hat.

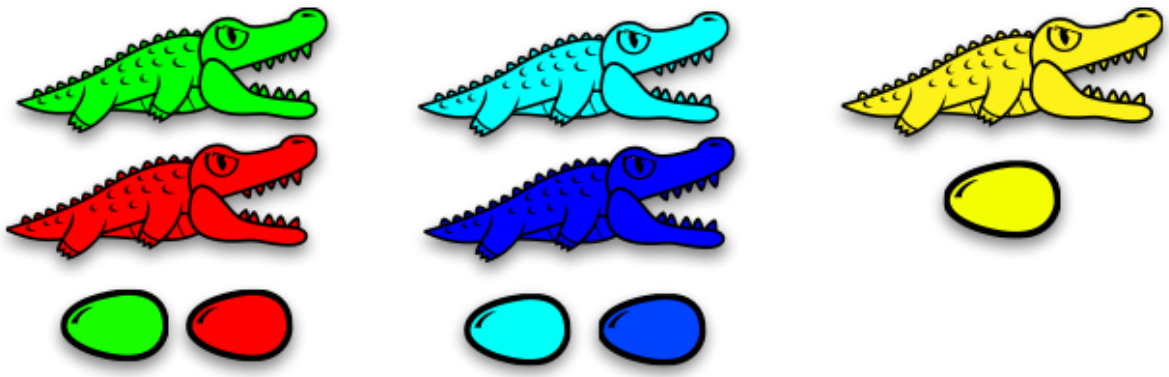


Die Farbenregel



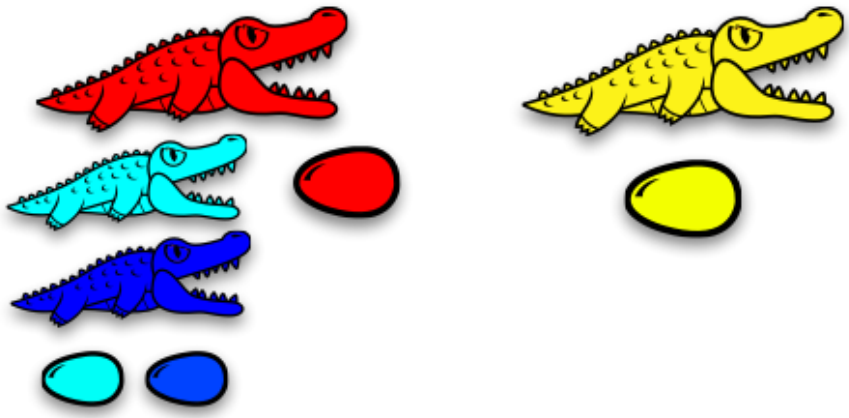
Setzen wir das Beispiel fort, frisst Orange Gelb und wir verbleiben mit dieser Konstellation. Jetzt gibt es allerdings ein Problem.

Die Farbenregel



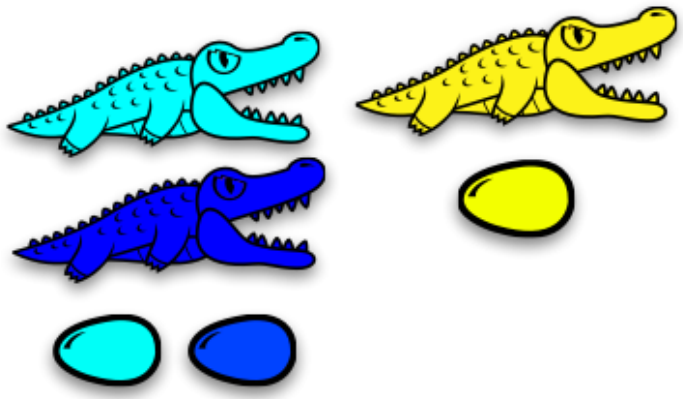
Setzen wir das Beispiel fort, frisst Orange Gelb und wir verbleiben mit dieser Konstellation. Jetzt gibt es allerdings ein Problem. Bevor ein Alligator eine Familie essen kann, in der eine Farbe vorkommt, die auch eins seiner Familienmitglieder hat, müssen die Farben geändert werden.

Die Farbenregel



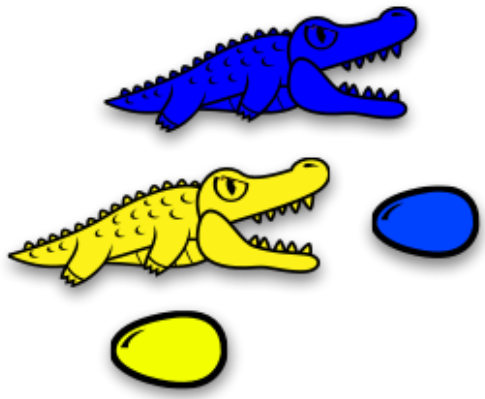
Setzen wir das Beispiel fort, frisst Orange Gelb und wir verbleiben mit dieser Konstellation. Jetzt gibt es allerdings ein Problem. Bevor ein Alligator eine Familie essen kann, in der eine Farbe vorkommt, die auch eins seiner Familienmitglieder hat, müssen die Farben geändert werden.
Dann können wir essen...

Die Farbenregel



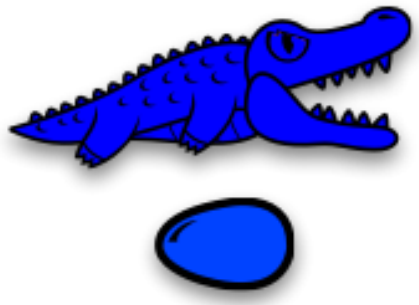
Setzen wir das Beispiel fort, frisst Orange Gelb und wir verbleiben mit dieser Konstellation. Jetzt gibt es allerdings ein Problem.
Bevor ein Alligator eine Familie essen kann, in der eine Farbe vorkommt, die auch eins seiner Familienmitglieder hat, müssen die Farben geändert werden.
Dann können wir essen und essen. . .

Die Farbenregel



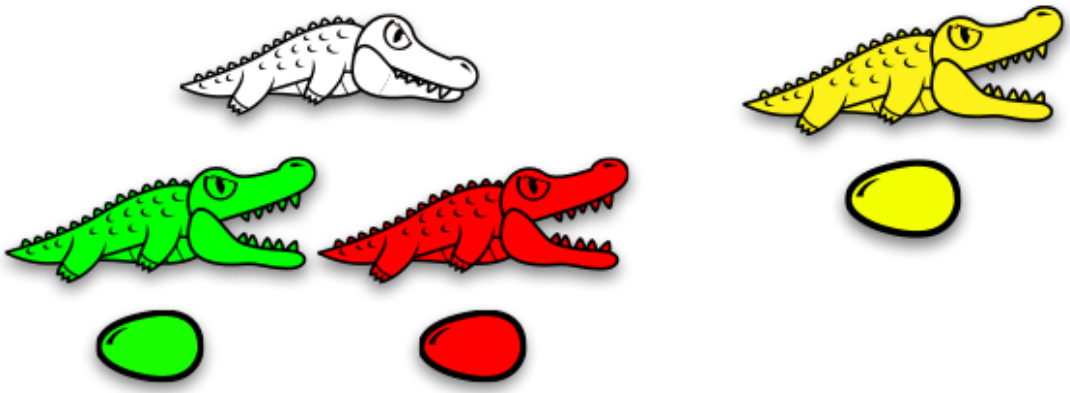
Setzen wir das Beispiel fort, frisst Orange Gelb und wir verbleiben mit dieser Konstellation. Jetzt gibt es allerdings ein Problem. Bevor ein Alligator eine Familie essen kann, in der eine Farbe vorkommt, die auch eins seiner Familienmitglieder hat, müssen die Farben geändert werden.
Dann können wir essen und essen und essen. . .

Die Farbenregel



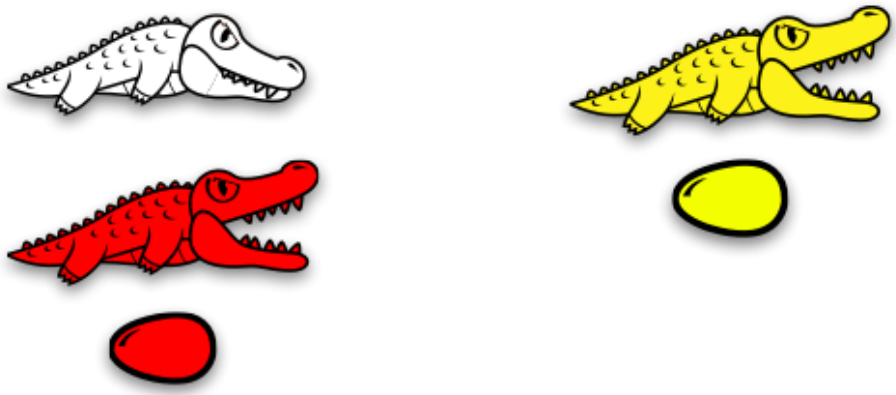
Setzen wir das Beispiel fort, frisst Orange Gelb und wir verbleiben mit dieser Konstellation. Jetzt gibt es allerdings ein Problem. Bevor ein Alligator eine Familie essen kann, in der eine Farbe vorkommt, die auch eins seiner Familienmitglieder hat, müssen die Farben geändert werden.
Dann können wir essen und essen und essen, bis alles weg ist.

Die Altersschwächeregel



Es gibt noch eine weitere Regel, die wir beachten müssen. Sie betrifft alte Alligatoren, die nicht mehr hungrig sind und nur noch ihre Familie bewachen (wie hier oben links). Unter welchen Bedingungen sterben diese?

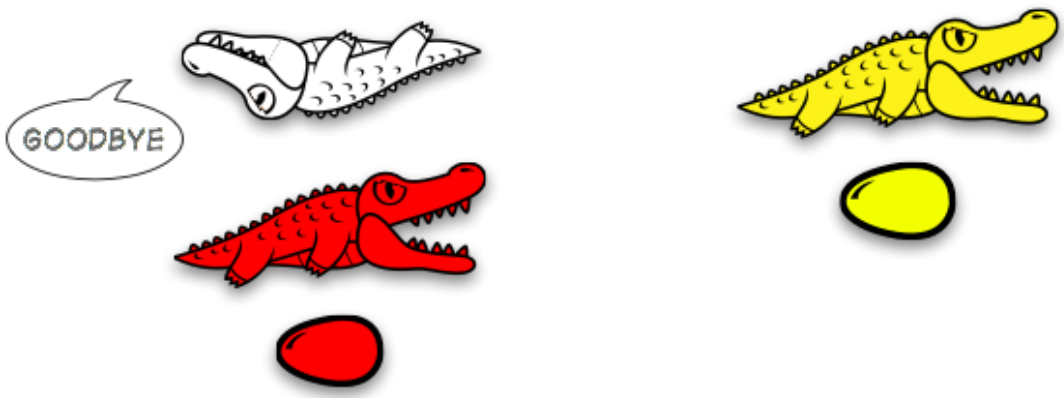
Die Altersschwächeregel



Die Antwort ist, dass sie sterben, wenn sie nur noch eine Familie beschützen.

Hier frisst der grüne Alligator die rote Familie und stirbt. Danach schlüpft eine neue rote Familie aus dem grünen Ei. Der alte Alligator bewacht jetzt nur noch eine Familie, die auch auf sich allein aufpassen kann. Er wird nicht mehr gebraucht, also stirbt er.

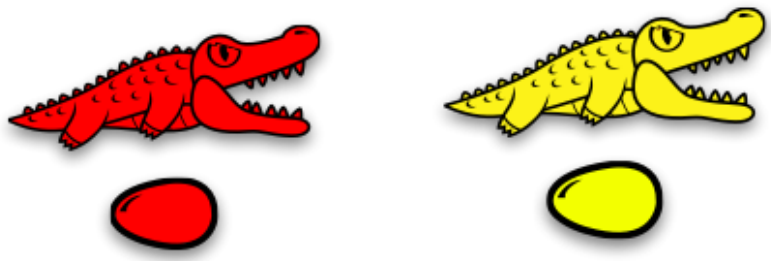
Die Altersschwächeregel



Die Antwort ist, dass sie sterben, wenn sie nur noch eine Familie beschützen.

Hier frisst der grüne Alligator die rote Familie und stirbt. Danach schlüpft eine neue rote Familie aus dem grünen Ei. Der alte Alligator bewacht jetzt nur noch eine Familie, die auch auf sich allein aufpassen kann. Er wird nicht mehr gebraucht, also stirbt er.

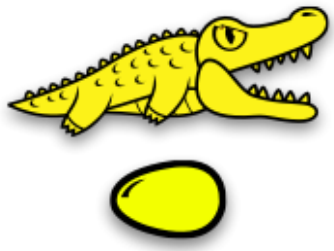
Die Altersschwächeregel



Die Antwort ist, dass sie sterben, wenn sie nur noch eine Familie beschützen.

Hier frisst der grüne Alligator die rote Familie und stirbt. Danach schlüpft eine neue rote Familie aus dem grünen Ei. Der alte Alligator bewacht jetzt nur noch eine Familie, die auch auf sich allein aufpassen kann. Er wird nicht mehr gebraucht, also stirbt er. Danach rückt der rote hungrige Alligator nach und frisst.

Die Altersschwächeregel



Die Antwort ist, dass sie sterben, wenn sie nur noch eine Familie beschützen.

Hier frisst der grüne Alligator die rote Familie und stirbt. Danach schlüpft eine neue rote Familie aus dem grünen Ei. Der alte Alligator bewacht jetzt nur noch eine Familie, die auch auf sich allein aufpassen kann. Er wird nicht mehr gebraucht, also stirbt er. Danach rückt der rote hungrige Alligator nach und frisst. Und so weiter und so fort, bis nichts mehr da ist.

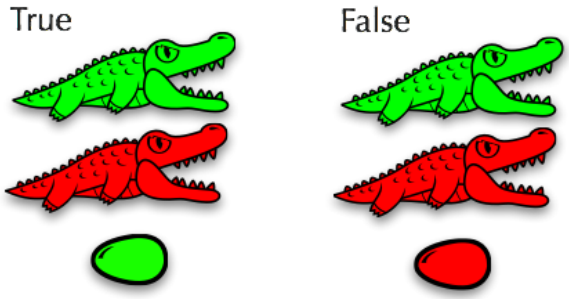
Gameplay

Das Ziel des Spiels ist nun, eine Familie auszuknobeln, die, wenn sie X gefüttert bekommt, Y produziert. Wir betrachten dabei zwei Familien unterschiedlicher Farben als „äquivalent“, wenn sie das gleiche Muster haben. Ein Beispiel:

Gameplay

Das Ziel des Spiels ist nun, eine Familie auszuknobeln, die, wenn sie X gefüttert bekommt, Y produziert. Wir betrachten dabei zwei Familien unterschiedlicher Farben als „äquivalent“, wenn sie das gleiche Muster haben. Ein Beispiel:

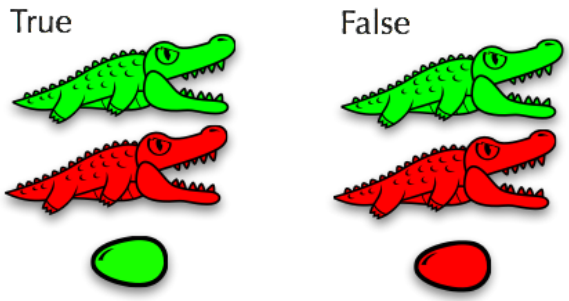
Hier sind zwei Familien, die wir „`True`“und „`False`“ nennen:



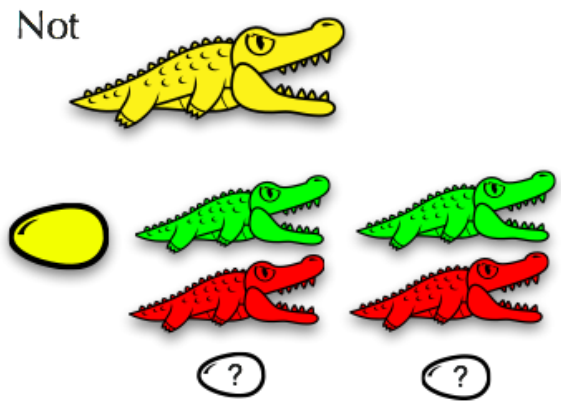
Gameplay

Das Ziel des Spiels ist nun, eine Familie auszuknobeln, die, wenn sie X gefüttert bekommt, Y produziert. Wir betrachten dabei zwei Familien unterschiedlicher Farben als „äquivalent“, wenn sie das gleiche Muster haben. Ein Beispiel:

Hier sind zwei Familien, die wir „`True`“und „`False`“ nennen:



Und hier ist die Familie „`Not`“:



Frage: Welche Farbe müssten die Eier der Familie `Not` haben?

Eine kleine Aufgabe, die eigentlich auf einem Übungszettel hätte landen sollen:

Eine kleine Aufgabe, die eigentlich auf einem Übungszettel hätte landen sollen:

Implementieren Sie eine kleine Alligator-DSL (Domain Specific Language) in `Haskell`, die einen gegebenen „Alligator-Ausdruck“ (grafisch?) auswertet.

Implementieren Sie außerdem die Konnektive `AND`, `OR`, `NAND` und `XOR` als Alligator-Ausdrücke.

λ-Kalkül & λ-Würfel

Was ist ein Lambdakalkül (engl. λ-calculus) und warum interessiert mich das?

Was ist ein Lambdakalkül (engl. λ-calculus) und warum interessiert mich das?

Das Lambdakalkül wurde in den 1930ern von Alonzo Church entworfen und formalisiert das Konzept „Berechnung“. Man könnte sagen, es ist die kleinste universelle Programmiersprache.
Heute noch spielt sie große Rollen in Programmiersprachentheorie.
`Haskell` z.B. basiert auf dem Lambdakalkül.

Empfehlung: Philipp Wadler (Prof. in Edinburgh) stand-up comedy zu computability theory:
`https://www.youtube.com/watch?v=GnpcMCW0RUA`

Was ist ein Lambdakalkül (engl. λ-calculus) und warum interessiert mich das?

Das Lambdakalkül wurde in den 1930ern von Alonzo Church entworfen und formalisiert das Konzept „Berechnung“. Man könnte sagen, es ist die kleinste universelle Programmiersprache.
Heute noch spielt sie große Rollen in Programmiersprachentheorie.
`Haskell` z.B. basiert auf dem Lambdakalkül.

Empfehlung: Philipp Wadler (Prof. in Edinburgh) stand-up comedy zu computability theory:
`https://www.youtube.com/watch?v=GnpcMCW0RUA`

. . . und lustigerweise ist all das äquivalent zu unseren Alligator-Ausdrücken!

Lambda-Calculus

Das Lambdakalkül setzt sich aus zwei Dingen zusammen: gültige *Terme* und *Umformungsregeln* zwischen Termen.

Ein gültiger Term ist eins von drei Dingen:

Lambda-Calculus

Das Lambdakalkül setzt sich aus zwei Dingen zusammen: gültige *Terme* und *Umformungsregeln* zwischen Termen.

Ein gültiger Term ist eins von drei Dingen:

- Eine Variable, (z.B.: $a, b, c, \ldots$)

Lambda-Calculus

Das Lambdakalkül setzt sich aus zwei Dingen zusammen: gültige *Terme* und *Umformungsregeln* zwischen Termen.

Ein gültiger Term ist eins von drei Dingen:

- Eine Variable, (z.B.: $a, b, c, \ldots$)
- Eine Lambda-Abstraktion (z.B.: $\lambda x.xx$)

Lambda-Calculus

Das Lambdakalkül setzt sich aus zwei Dingen zusammen: gültige *Terme* und *Umformungsregeln* zwischen Termen.

Ein gültiger Term ist eins von drei Dingen:

- Eine Variable, (z.B.: $a, b, c, \ldots$)
- Eine Lambda-Abstraktion (z.B.: $\lambda x.xx$)
- Eine Anwendung eines Terms auf einen anderen (z.B.: $(\lambda x.xx)(palim)$)

α-Konversion

Die erste Konversionsregel wird α-Konversion, oder Umbenennung genannt:

$$\lambda V.E \equiv \lambda W.E[V \leftarrow W]$$

α-Konversion

Die erste Konversionsregel wird α-Konversion, oder Umbenennung genannt:

$$\lambda V.E \equiv \lambda W.E[V \leftarrow W]$$

Diese Regel besagt quasi, dass Variablennamen irrelephant sind. $(\lambda x.xx)$ und $(\lambda y.yy)$ sind diesselbe Funktion. Die Details sind allerdings etwas komplizierter, wir werden hier nicht näher drauf eingehen.

α-Konversion

Die erste Konversionsregel wird α-Konversion, oder Umbenennung genannt:

$$\lambda V.E \equiv \lambda W.E[V \leftarrow W]$$

Diese Regel besagt quasi, dass Variablennamen irrelephant sind. $(\lambda x.xx)$ und $(\lambda y.yy)$ sind diesselbe Funktion. Die Details sind allerdings etwas komplizierter, wir werden hier nicht näher drauf eingehen.
Können zwei Terme ineinander umgewandelt werden, nennt man sie α-äquivalent. Meistens werden α-äquivalente Terme als identisch angesehen.

β-Reduktion

Die zweite Konversionsregel, genannt β-Reduktion, bildet das Konzept von Funktionsanwendung ab:

$$(\lambda V.E)E' \equiv E[V \leftarrow E']$$

β-Reduktion

Die zweite Konversionsregel, genannt β-Reduktion, bildet das Konzept von Funktionsanwendung ab:

$$(\lambda V.E)E' \equiv E[V \leftarrow E']$$

Hier wird einfach das Argument an die Stelle der Variablen eingesetzt. Eine Bedingung ist allerdings, dass alle freien Variablen frei bleiben müssen.

β-Reduktion

Die zweite Konversionsregel, genannt β-Reduktion, bildet das Konzept von Funktionsanwendung ab:

$$(\lambda V.E)E' \equiv E[V \leftarrow E']$$

Hier wird einfach das Argument an die Stelle der Variablen eingesetzt. Eine Bedingung ist allerdings, dass alle freien Variablen frei bleiben müssen.
Terme, auf die diese Regel angewandt werden kann, heißen β-reduzibel. Allerdings haben nicht alle Terme auch eine β-Normalform ($(\lambda x.xx)(\lambda x.xx)$ z.B. ist reduzibel, ergibt sich aber selbst).

η-Konversion

Die dritte Konversionsregel, genannt η-Konversion, ist lediglich optional:

$$\lambda x.fx \equiv f$$

η-Konversion

Die dritte Konversionsregel, genannt η-Konversion, ist lediglich optional:

$$\lambda x.fx \equiv f$$

Bedingung ist, dass x keine freie Variable von f ist. Diese Regel kann hinzugefügt werden, wenn man *Extensionalität* abbilden möchte. Das bedeutet, dass zwei Funktionen gleich sind, wenn sie für alle Eingaben das gleiche Ergebnis liefern.

η-Konversion

Die dritte Konversionsregel, genannt η-Konversion, ist lediglich optional:

$$\lambda x.fx \equiv f$$

Bedingung ist, dass x keine freie Variable von f ist. Diese Regel kann hinzugefügt werden, wenn man *Extensionalität* abbilden möchte. Das bedeutet, dass zwei Funktionen gleich sind, wenn sie für alle Eingaben das gleiche Ergebnis liefern.

Beispiel: `quicksort` und `heapsort` sind *extensional* gleich ($\forall x.\ \texttt{mergesort}(x) = \texttt{heapsort}(x)$), *intensional* allerdings nicht.

Lambda calculus-Alligator-Isomorphism

Wie vorhin versprochen gibt es eine genaue Übersetzung von unseren Alligatoren zum Lambda-Kalkül und umgekehrt:

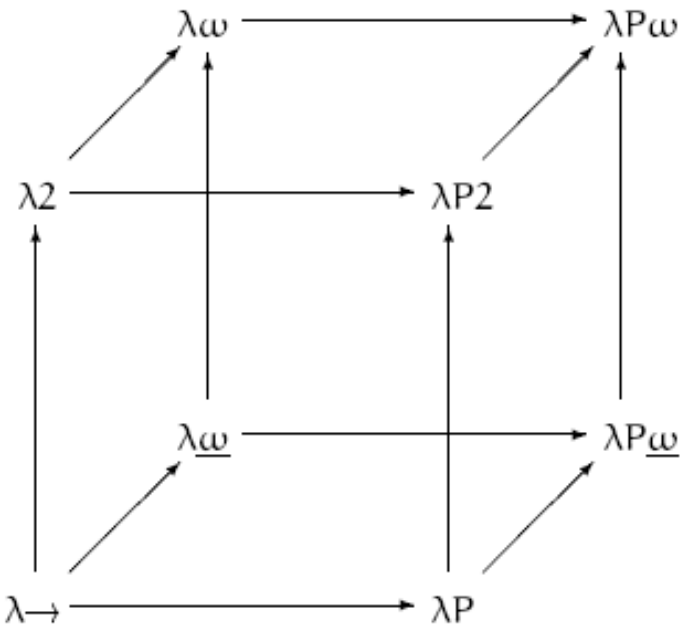
Lambda calculus-Alligator-Isomorphism

Wie vorhin versprochen gibt es eine genaue Übersetzung von unseren Alligatoren zum Lambda-Kalkül und umgekehrt:

Alligatoren	**Lambda-Kalkül**
Hungriger Alligator	Lambda-Abstraktion
Eier	Variablen
Alter Alligator	Klammern
Essensregel	β-Reduktion
Farbenregel	α-Konversion
Altersregel	Klammern löschen

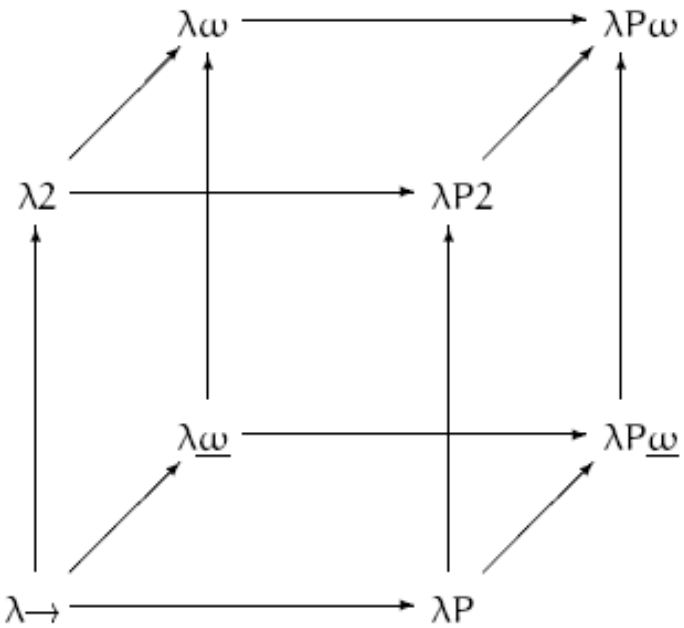
Das Lambdakalkül mit Typen (eine Einschränkung des untypisierten Kalküls) ist allerdings nur der Anfang. Im so genannten „Lambda-Würfel“ ist es der Ursprung ($\lambda \rightarrow$).

Das Lambdakalkül mit Typen (eine Einschränkung des untypisierten Kalküls) ist allerdings nur der Anfang. Im so genannten „Lambda-Würfel“ ist es der Ursprung ($\lambda \rightarrow$).



Im Würfel gibt es drei Richtungen der Abstraktion:

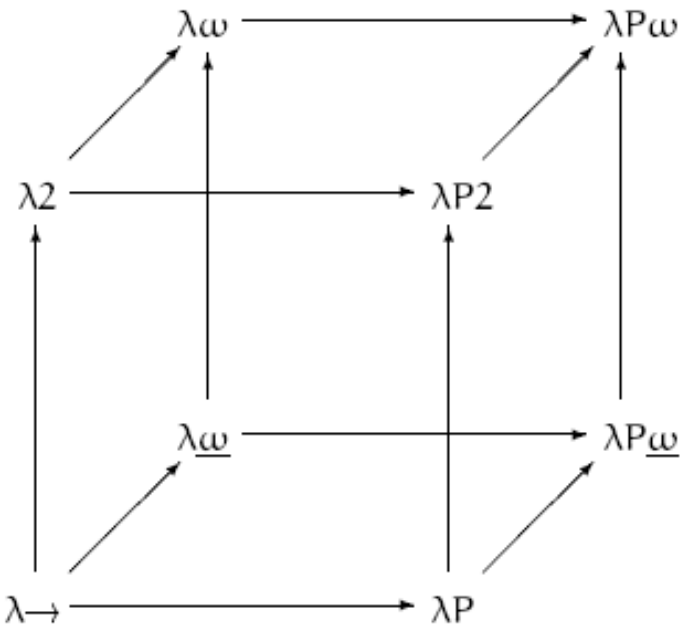
Das Lambdakalkül mit Typen (eine Einschränkung des untypisierten Kalküls) ist allerdings nur der Anfang. Im so genannten „Lambda-Würfel“ ist es der Ursprung ($\lambda \rightarrow$).



Im Würfel gibt es drei Richtungen der Abstraktion:

- *Terms depending on types*
  Auch „Polymorphismus“

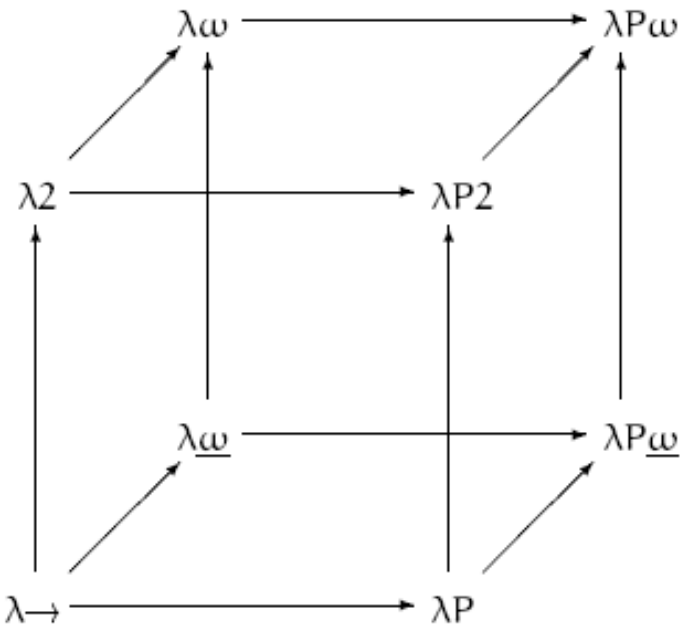
Das Lambdakalkül mit Typen (eine Einschränkung des untypisierten Kalküls) ist allerdings nur der Anfang. Im so genannten „Lambda-Würfel“ ist es der Ursprung ($\lambda \rightarrow$).



Im Würfel gibt es drei Richtungen der Abstraktion:

- *Terms depending on types*
  Auch „Polymorphismus“
- *Types depending on types*
  Auch „Type Operators“

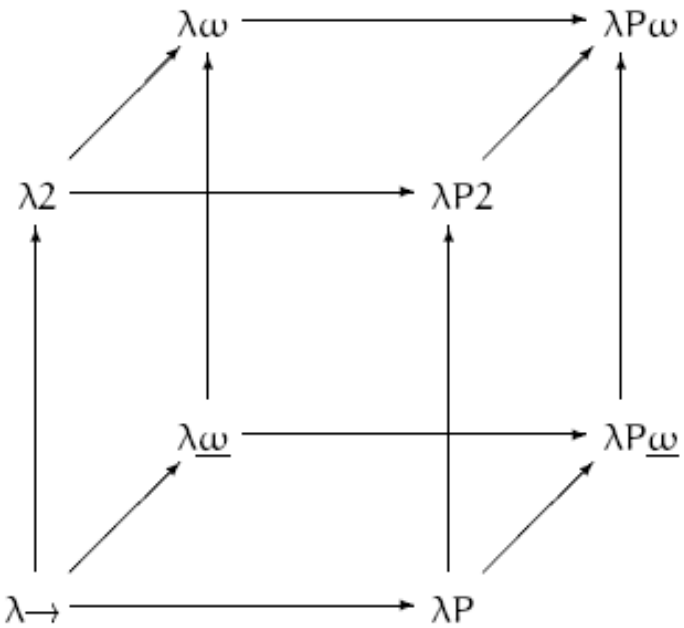
Das Lambdakalkül mit Typen (eine Einschränkung des untypisierten Kalküls) ist allerdings nur der Anfang. Im so genannten „Lambda-Würfel“ ist es der Ursprung ($\lambda \rightarrow$).



Im Würfel gibt es drei Richtungen der Abstraktion:

- *Terms depending on types*
  Auch „Polymorphismus“
- *Types depending on types*
  Auch „Type Operators“
- *Types depending on terms*
  Auch „Dependent Types“

Das Lambdakalkül mit Typen (eine Einschränkung des untypisierten Kalküls) ist allerdings nur der Anfang. Im so genannten „Lambda-Würfel“ ist es der Ursprung ($\lambda \rightarrow$).



Im Würfel gibt es drei Richtungen der Abstraktion:

- *Terms depending on types*
  Auch „Polymorphismus“
- *Types depending on types*
  Auch „Type Operators“
- *Types depending on terms*
  Auch „Dependent Types“

Das obere Ende des Würfels, $\lambda\Pi\omega$, ist auch als „calculus of constructions“ bekannt (entwickelt von Thierry Coquand) und dient als Basis für den Beweisassistenten `Coq`.

```
(++) : Vect m a -> Vect n a -> Vect (m + n) a
(++) []       ys = ys
(++) (x::xs) ys = x :: xs ++ ys
```

Back to the roots:

Monads & Categories

Zum Abschluss wollen wir noch ein besonders bekanntes Meme der Haskell-Community genauer unter die Lupe nehmen. Den Satz:

„Monaden sind Monoide in der Kategorie der Endofunktoren."

Zum Abschluss wollen wir noch ein besonders bekanntes Meme der Haskell-Community genauer unter die Lupe nehmen. Den Satz:

> „*Monaden sind Monoide in der Kategorie der Endofunktoren.*“

. . . dafür werden wir allerdings einen kleinen Abstecher in die Mathematik benötigen.

Eine kurze Erinnerung:

`Monad` ist eine Typklasse in `Haskell`, die die Implementation von `return` und »= (Bind) voraussetzt. Insbesondere ist jeder Typ in `Monad` ebenfalls in `Applicative` und `Functor`.

```
class Applicative m => Monad m where
    return :: a -> m a
    (>>=)  :: m a -> (a -> m b) -> m b
```

Eine kurze Erinnerung:

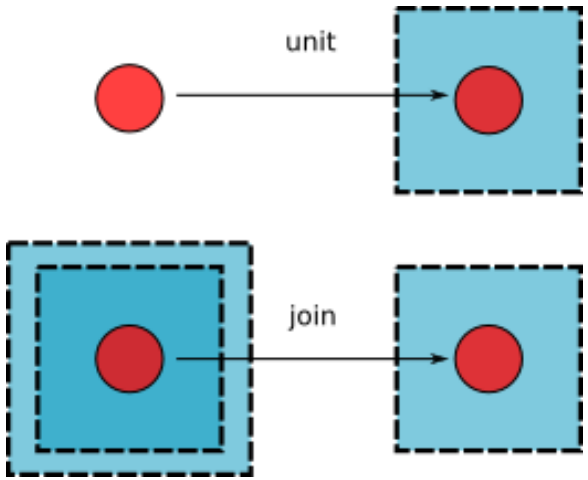
`Monad` ist eine Typklasse in `Haskell`, die die Implementation von `return` und »= (Bind) voraussetzt. Insbesondere ist jeder Typ in `Monad` ebenfalls in `Applicative` und `Functor`.

```
class Applicative m => Monad m where
    return :: a -> m a
    (>>=)  :: m a -> (a -> m b) -> m b
```

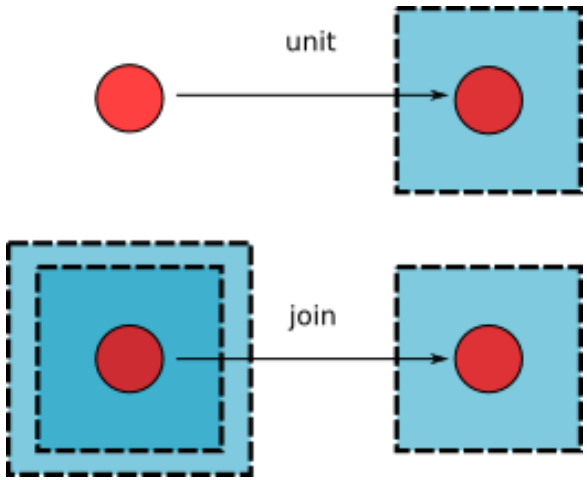
Besonders beliebt: `Identity`, `[]`, `IO`, `Maybe`...

Statt `return` und `(>>=)` kann man auch andere Funktionen verwenden. In der Kategorientheorie werden häufig $\eta : I \to M$ (unit) und $\mu : M \times M \to M$ (join) verwendet.

Statt `return` und `(>>=)` kann man auch andere Funktionen verwenden. In der Kategorientheorie werden häufig $\eta : I \to M$ (unit) und $\mu : M \times M \to M$ (join) verwendet.



Statt `return` und `(>>=)` kann man auch andere Funktionen verwenden. In der Kategorientheorie werden häufig $\eta : I \to M$ (unit) und $\mu : M \times M \to M$ (join) verwendet.




```haskell
unit :: Monad m -> a -> m a
unit = return

join :: Monad m => m (m a)
                -> m a
join x = x >>= id

(>>=) :: Monad m => m a
                 -> (a -> m b)
                 -> m b
x >>= f = join (fmap f x)
```

Definition (Algebra):

Sei M eine Menge und $\circ$ eine (abgeschl.) binäre Relation auf M (d.h. $\circ : M \times M \to M$). Dann heißt $(M, \circ)$ *Monoid*, wenn $\circ$ die folgenden zwei Axiome beachtet:

Definition (Algebra):

Sei M eine Menge und $\circ$ eine (abgeschl.) binäre Relation auf M (d.h. $\circ : M \times M \to M$). Dann heißt $(M, \circ)$ *Monoid*, wenn $\circ$ die folgenden zwei Axiome beachtet:

- (ass) $\forall a, b, c \in M : \ (a \circ b) \circ c = a \circ (b \circ c)$

Definition (Algebra):

Sei M eine Menge und $\circ$ eine (abgeschl.) binäre Relation auf M (d.h. $\circ : M \times M \to M$). Dann heißt $(M, \circ)$ *Monoid*, wenn $\circ$ die folgenden zwei Axiome beachtet:

- (ass) $\forall a, b, c \in M : \ (a \circ b) \circ c = a \circ (b \circ c)$
- (ide) $\exists e \in M$ sodass $\forall a \in m : \ e \circ a = a \circ e = a$

Definition (Algebra):

Sei M eine Menge und $\circ$ eine (abgeschl.) binäre Relation auf M (d.h. $\circ : M \times M \to M$). Dann heißt $(M, \circ)$ *Monoid*, wenn $\circ$ die folgenden zwei Axiome beachtet:

- (ass) $\forall a, b, c \in M : \ (a \circ b) \circ c = a \circ (b \circ c)$
- (ide) $\exists e \in M$ sodass $\forall a \in m : \ e \circ a = a \circ e = a$

Beliebte Monoide:

Definition (Algebra):

Sei M eine Menge und $\circ$ eine (abgeschl.) binäre Relation auf M (d.h. $\circ : M \times M \to M$). Dann heißt $(M, \circ)$ *Monoid*, wenn $\circ$ die folgenden zwei Axiome beachtet:

- (ass) $\forall a, b, c \in M : \ (a \circ b) \circ c = a \circ (b \circ c)$
- (ide) $\exists e \in M$ sodass $\forall a \in m : \ e \circ a = a \circ e = a$

Beliebte Monoide:

- $(\mathbb{N}, +), (\mathbb{N}, \cdot), (\{\bot, \top\}, \texttt{AND}), (\{\bot, \top\}, \texttt{OR}) \ldots$

Definition (Algebra):

Sei M eine Menge und $\circ$ eine (abgeschl.) binäre Relation auf M (d.h. $\circ : M \times M \to M$). Dann heißt $(M, \circ)$ *Monoid*, wenn $\circ$ die folgenden zwei Axiome beachtet:

- (ass) $\forall a, b, c \in M : \ (a \circ b) \circ c = a \circ (b \circ c)$
- (ide) $\exists e \in M$ sodass $\forall a \in m : \ e \circ a = a \circ e = a$

Beliebte Monoide:

- $(\mathbb{N}, +), (\mathbb{N}, \cdot), (\{\bot, \top\}, \texttt{AND}), (\{\bot, \top\}, \texttt{OR}) \ldots$
- Gegeben eine Menge A, $\mathcal{P}(A)$ entweder mit $\cap$ oder $\cup$

Definition (Algebra):

Sei M eine Menge und $\circ$ eine (abgeschl.) binäre Relation auf M (d.h. $\circ : M \times M \to M$). Dann heißt $(M, \circ)$ *Monoid*, wenn $\circ$ die folgenden zwei Axiome beachtet:

- (ass) $\forall a, b, c \in M : \ (a \circ b) \circ c = a \circ (b \circ c)$
- (ide) $\exists e \in M$ sodass $\forall a \in m : \ e \circ a = a \circ e = a$

Beliebte Monoide:

- $(\mathbb{N}, +), (\mathbb{N}, \cdot), (\{\bot, \top\}, \texttt{AND}), (\{\bot, \top\}, \texttt{OR}) \ldots$
- Gegeben eine Menge A, $\mathcal{P}(A)$ entweder mit $\cap$ oder $\cup$
- Jede Menge mit nur einem Element formt einen trivialen Monoid mit der trivialen Relation.

Kategorientheorie

Worum geht es bei Kategorientheorie?

Kategorientheorie

Worum geht es bei Kategorientheorie?

Kategorientheorie kann als die „Mathematik der Mathematik“ betrachtet werden. Es geht um eine Abstraktion vieler Konzepte aus unterschiedlichen Zweigen der Mathematik und die Aufdeckung fundamentaler Ähnlichkeiten zwischen ihnen.

Kategorientheorie

Worum geht es bei Kategorientheorie?

Kategorientheorie kann als die „Mathematik der Mathematik“ betrachtet werden. Es geht um eine Abstraktion vieler Konzepte aus unterschiedlichen Zweigen der Mathematik und die Aufdeckung fundamentaler Ähnlichkeiten zwischen ihnen.

„Category theory has been labelled as 'generalised abstract nonsense' by both its critics and its proponents!“

Kategorientheorie

Eine Kategorie, oft genannt $\mathcal{C}$, besteht aus:

Kategorientheorie

Eine Kategorie, oft genannt $\mathcal{C}$, besteht aus:

- Einer Klasse (engl. „collection“, keine Menge!) aus *Objekten*, genannt $Obj(\mathcal{C})$.

Kategorientheorie

Eine Kategorie, oft genannt $\mathcal{C}$, besteht aus:

- Einer Klasse (engl. „collection“, keine Menge!) aus *Objekten*, genannt $Obj(\mathcal{C})$.
- Einer Klasse von *Pfeilen* (oder auch *Morphismen*) zwischen Objekten, genannt $Hom(\mathcal{C})$. Ein Morphismus in $\mathcal{C}$ von Element X zu Element Y wird oft als $Hom_{\mathcal{C}}(X, Y)$ notiert.

Kategorientheorie

Eine Kategorie, oft genannt $\mathcal{C}$, besteht aus:

- Einer Klasse (engl. „collection“, keine Menge!) aus *Objekten*, genannt $Obj(\mathcal{C})$.
- Einer Klasse von *Pfeilen* (oder auch *Morphismen*) zwischen Objekten, genannt $Hom(\mathcal{C})$. Ein Morphismus in $\mathcal{C}$ von Element X zu Element Y wird oft als $Hom_{\mathcal{C}}(X, Y)$ notiert.
- Einer assoziativen Verknüpfungsrelation für Morphismen:

$$Hom_{\mathcal{C}}(Y, Z) \times Hom_{\mathcal{C}}(X, Y) \to Hom_{\mathcal{C}}(X, Z)$$

$$(g, f) \mapsto g \circ f$$

Kategorientheorie

Eine Kategorie, oft genannt $\mathcal{C}$, besteht aus:

- Einer Klasse (engl. „collection“, keine Menge!) aus *Objekten*, genannt $Obj(\mathcal{C})$.
- Einer Klasse von *Pfeilen* (oder auch *Morphismen*) zwischen Objekten, genannt $Hom(\mathcal{C})$. Ein Morphismus in $\mathcal{C}$ von Element X zu Element Y wird oft als $Hom_{\mathcal{C}}(X, Y)$ notiert.
- Einer assoziativen Verknüpfungsrelation für Morphismen:

$$Hom_{\mathcal{C}}(Y, Z) \times Hom_{\mathcal{C}}(X, Y) \to Hom_{\mathcal{C}}(X, Z)$$

$$(g, f) \mapsto g \circ f$$

- Einem Identitätsmorphismus für jedes Objekt X, notiert als id_X, der als neutrales Element für die Verknüpfung von Morphismen mit Domäne oder Codomäne X fungiert.

Funktoren

Und was ist mit Funktoren?

Funktoren

Und was ist mit Funktoren?

Seien $\mathcal{C}$ und $\mathcal{D}$ Kategorien. Ein *Funktor* F ist eine Abbildung von $\mathcal{C}$ nach $\mathcal{D}$, die folgende Bedingungen einhält:

Funktoren

Und was ist mit Funktoren?

Seien $\mathcal{C}$ und $\mathcal{D}$ Kategorien. Ein *Funktor* F ist eine Abbildung von $\mathcal{C}$ nach $\mathcal{D}$, die folgende Bedingungen einhält:

- F bildet jedes $X \in Obj(\mathcal{C})$ auf ein $F(X) \in Obj(\mathcal{D})$ ab.

Funktoren

Und was ist mit Funktoren?

Seien $\mathcal{C}$ und $\mathcal{D}$ Kategorien. Ein *Funktor* F ist eine Abbildung von $\mathcal{C}$ nach $\mathcal{D}$, die folgende Bedingungen einhält:

- F bildet jedes $X \in Obj(\mathcal{C})$ auf ein $F(X) \in Obj(\mathcal{D})$ ab.
- F bildet jedes $f \in Hom(\mathcal{C})$ auf ein $F(f) \in Hom(\mathcal{D})$ ab, sodass folgendes gilt:

Funktoren

Und was ist mit Funktoren?

Seien $\mathcal{C}$ und $\mathcal{D}$ Kategorien. Ein *Funktor* F ist eine Abbildung von $\mathcal{C}$ nach $\mathcal{D}$, die folgende Bedingungen einhält:

- F bildet jedes $X \in Obj(\mathcal{C})$ auf ein $F(X) \in Obj(\mathcal{D})$ ab.
- F bildet jedes $f \in Hom(\mathcal{C})$ auf ein $F(f) \in Hom(\mathcal{D})$ ab, sodass folgendes gilt:
  - $F(id_X) = id_{F(X)}$

Funktoren

Und was ist mit Funktoren?

Seien $\mathcal{C}$ und $\mathcal{D}$ Kategorien. Ein *Funktor* F ist eine Abbildung von $\mathcal{C}$ nach $\mathcal{D}$, die folgende Bedingungen einhält:

- F bildet jedes $X \in Obj(\mathcal{C})$ auf ein $F(X) \in Obj(\mathcal{D})$ ab.
- F bildet jedes $f \in Hom(\mathcal{C})$ auf ein $F(f) \in Hom(\mathcal{D})$ ab, sodass folgendes gilt:
  - $F(id_X) = id_{F(X)}$
  - $F(g \circ f) = F(g) \circ F(f)\ \forall f : X \to Y, g : Y \to Z$

Funktoren

Und was ist mit Funktoren?

Seien $\mathcal{C}$ und $\mathcal{D}$ Kategorien. Ein *Funktor* F ist eine Abbildung von $\mathcal{C}$ nach $\mathcal{D}$, die folgende Bedingungen einhält:

- F bildet jedes $X \in Obj(\mathcal{C})$ auf ein $F(X) \in Obj(\mathcal{D})$ ab.
- F bildet jedes $f \in Hom(\mathcal{C})$ auf ein $F(f) \in Hom(\mathcal{D})$ ab, sodass folgendes gilt:
  - $F(id_X) = id_{F(X)}$
  - $F(g \circ f) = F(g) \circ F(f)\ \forall f : X \to Y, g : Y \to Z$

Man sagt auch oft, dass Funktoren wegen ihrer Eigenschaften *strukturerhaltend* sind.

Funktoren

Und was ist mit Funktoren?

Seien $\mathcal{C}$ und $\mathcal{D}$ Kategorien. Ein *Funktor* F ist eine Abbildung von $\mathcal{C}$ nach $\mathcal{D}$, die folgende Bedingungen einhält:

- F bildet jedes $X \in Obj(\mathcal{C})$ auf ein $F(X) \in Obj(\mathcal{D})$ ab.
- F bildet jedes $f \in Hom(\mathcal{C})$ auf ein $F(f) \in Hom(\mathcal{D})$ ab, sodass folgendes gilt:
  - $F(id_X) = id_{F(X)}$
  - $F(g \circ f) = F(g) \circ F(f)\ \forall f : X \to Y, g : Y \to Z$

Man sagt auch oft, dass Funktoren wegen ihrer Eigenschaften *strukturerhaltend* sind.

Eine Abbildung von einem Funktor F auf einen Funktor G heißt „natürliche Abbildung“!

Funktoren

Und was ist mit Funktoren?

Seien $\mathcal{C}$ und $\mathcal{D}$ Kategorien. Ein *Funktor* F ist eine Abbildung von $\mathcal{C}$ nach $\mathcal{D}$, die folgende Bedingungen einhält:

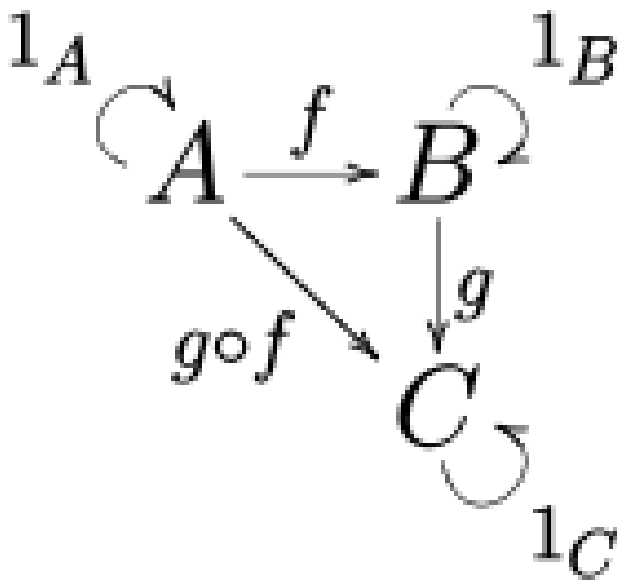
- F bildet jedes $X \in Obj(\mathcal{C})$ auf ein $F(X) \in Obj(\mathcal{D})$ ab.
- F bildet jedes $f \in Hom(\mathcal{C})$ auf ein $F(f) \in Hom(\mathcal{D})$ ab, sodass folgendes gilt:
  - $F(id_X) = id_{F(X)}$
  - $F(g \circ f) = F(g) \circ F(f)\ \forall f : X \to Y, g : Y \to Z$

Man sagt auch oft, dass Funktoren wegen ihrer Eigenschaften *strukturerhaltend* sind.

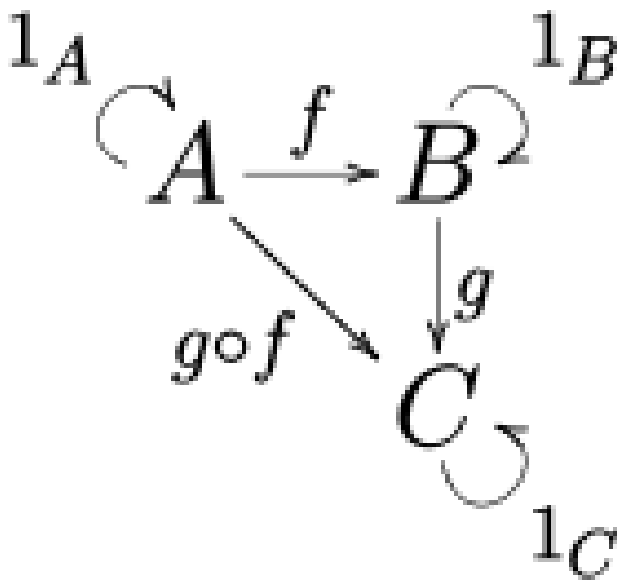
Eine Abbildung von einem Funktor F auf einen Funktor G heißt „natürliche Abbildung“!

Ein Funktor heißt *Endo*funktor, einfach wenn $\mathcal{C} = \mathcal{D}$.

Kategorientheorie

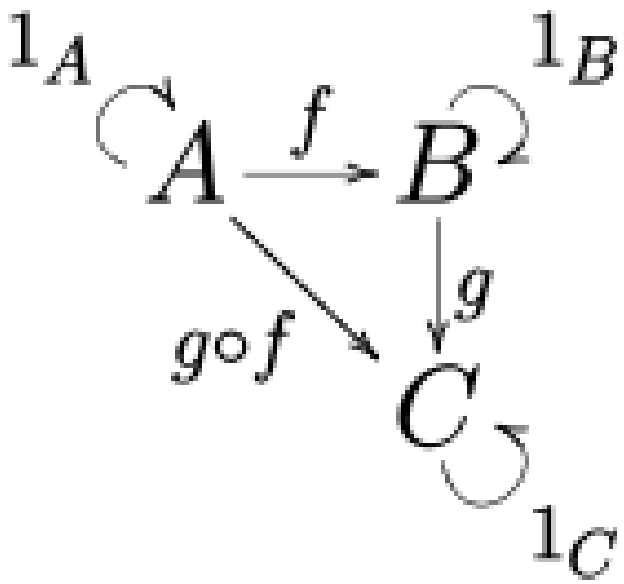


Kategorientheorie



Beliebte Kategorien:

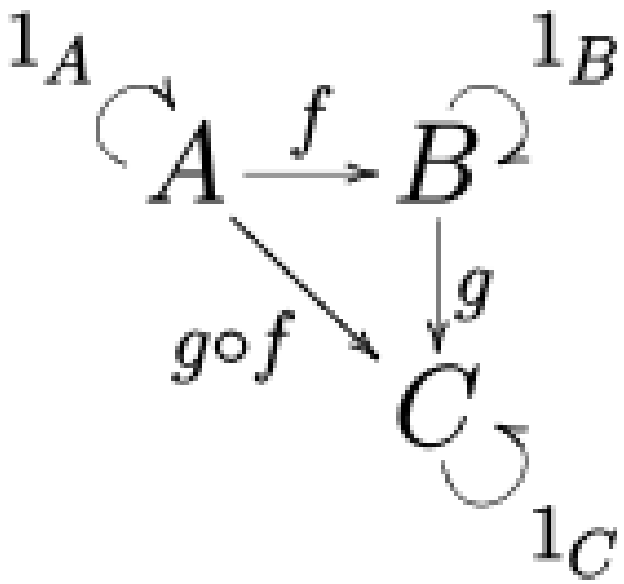
Kategorientheorie



Beliebte Kategorien:

- Die Kategorie **Set**, von Mengen und Funktionen

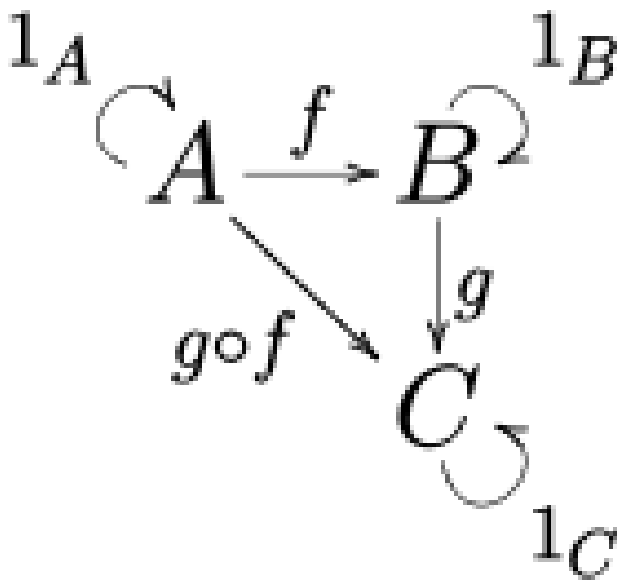
Kategorientheorie



Beliebte Kategorien:

- Die Kategorie **Set**, von Mengen und Funktionen
- Die Kategorie **Cat**, von Kategorien und Funktoren

Kategorientheorie



Beliebte Kategorien:

- Die Kategorie **Set**, von Mengen und Funktionen
- Die Kategorie **Cat**, von Kategorien und Funktoren
- Die Kategorie **Top**, von topologischen Räumen und stetigen Abbildungen

Fügen wir also zusammen, was wir gelernt haben:

Fügen wir also zusammen, was wir gelernt haben:

- Wir betrachten die Kategorie der Typen in `Haskell`: `Hask`.

Fügen wir also zusammen, was wir gelernt haben:

- Wir betrachten die Kategorie der Typen in `Haskell`: `Hask`.
- `Functors` (die Typklasse) bilden `Hask` auf `Hask` ab, sind also Endofunktoren.

Fügen wir also zusammen, was wir gelernt haben:

- Wir betrachten die Kategorie der Typen in `Haskell`: `Hask`.
- `Functors` (die Typklasse) bilden `Hask` auf `Hask` ab, sind also Endofunktoren.
- Nehmen wir einen dieser Endofunktoren als Basis eines Monoiden, brauchen wir noch ein neutrales Element und eine abgeschl. binäre Relation.

Fügen wir also zusammen, was wir gelernt haben:

- Wir betrachten die Kategorie der Typen in `Haskell`: `Hask`.
- `Functors` (die Typklasse) bilden `Hask` auf `Hask` ab, sind also Endofunktoren.
- Nehmen wir einen dieser Endofunktoren als Basis eines Monoiden, brauchen wir noch ein neutrales Element und eine abgeschl. binäre Relation.
- $\eta : I \to M$ (unit) und $\mu : M \times M \to M$ (join) sind genau das.

Fügen wir also zusammen, was wir gelernt haben:

- Wir betrachten die Kategorie der Typen in `Haskell`: `Hask`.
- `Functors` (die Typklasse) bilden `Hask` auf `Hask` ab, sind also Endofunktoren.
- Nehmen wir einen dieser Endofunktoren als Basis eines Monoiden, brauchen wir noch ein neutrales Element und eine abgeschl. binäre Relation.
- $\eta : I \to M$ (unit) und $\mu : M \times M \to M$ (join) sind genau das.
- Ein `Functor` mit μ und η ist eine Monade!

Fügen wir also zusammen, was wir gelernt haben:

- Wir betrachten die Kategorie der Typen in `Haskell`: `Hask`.
- `Functors` (die Typklasse) bilden `Hask` auf `Hask` ab, sind also Endofunktoren.
- Nehmen wir einen dieser Endofunktoren als Basis eines Monoiden, brauchen wir noch ein neutrales Element und eine abgeschl. binäre Relation.
- $\eta : I \to M$ (unit) und $\mu : M \times M \to M$ (join) sind genau das.
- Ein `Functor` mit μ und η ist eine Monade!

$\therefore$ „*Monaden sind Monoide in der Kategorie der Endofunktoren.*“


That's all Folks!