The Treasure's in the Details

Jacques Carette, Musa Al-hassy, Wolfram Kahl

McMaster University, Hamilton

July 19, 2018

https://github.com/JacquesCarette/TheoriesAndDataStructures
WORK IN PROGRESS

DeepSpec 2018, Princeton University

A Relationship Between Mathematics and Computing Science

The Curry-Howard Correspondence —"Propositions as Types"

Logic	**Programming**	Example Use in Programming
$true$	singleton type	return type of side-effect only methods
$false$	empty type	return type for non-terminating methods

A Relationship Between Mathematics and Computing Science

The Curry-Howard Correspondence —"Propositions as Types"

Logic	**Programming**		Example Use in Programming
$true$	singleton type		return type of side-effect only methods
$false$	empty type		return type for non-terminating methods
$\Rightarrow$	function type	$\rightarrow$	methods with an input and output type
$\wedge$	product type	$\times$	simple records of data and methods
$\vee$	sum type	$+$	enumerations or tagged unions

A Relationship Between Mathematics and Computing Science

The Curry-Howard Correspondence —"Propositions as Types"

Logic	**Programming**		Example Use in Programming
true	singleton type		return type of side-effect only methods
false	empty type		return type for non-terminating methods
$\Rightarrow$	function type	$\rightarrow$	methods with an input and output type
$\wedge$	product type	$\times$	simple records of data and methods
$\vee$	sum type	$+$	enumerations or tagged unions
$\forall$	dependent function type	Π	return type varies according to input *value*
$\exists$	dependent product type	Σ	record fields depend on each other's *values*

A Relationship Between Mathematics and Computing Science

The Curry-Howard Correspondence —"Propositions as Types"

Logic	**Programming**		Example Use in Programming
true	singleton type		return type of side-effect only methods
false	empty type		return type for non-terminating methods
$\Rightarrow$	function type	$\rightarrow$	methods with an input and output type
$\wedge$	product type	$\times$	simple records of data and methods
$\vee$	sum type	$+$	enumerations or tagged unions
$\forall$	dependent function type	Π	return type varies according to input *value*
$\exists$	dependent product type	Σ	record fields depend on each other's *values*
natural deduction	type system		ensuring only "meaningful" programs
hypothesis	free variables		global variables, closures
implication elimination	function application		executing methods on arguments
implication introduction	λ-abstraction		parameters acting as local variables to method definitions

A Relationship Between Mathematics and Computing Science

$\boxed{\textbf{\textit{Proposition}}}$ P **is either** $true$ **or** $false$**.**

The Curry-Howard Correspondence —"Propositions as Types"

Logic	Programming		Example Use in Programming
$true$	singleton type		return type of side-effect only methods
$false$	empty type		return type for non-terminating methods
$\Rightarrow$	function type	$\to$	methods with an input and output type
$\wedge$	product type	$\times$	simple records of data and methods
$\vee$	sum type	$+$	enumerations or tagged unions
$\forall$	dependent function type	Π	return type varies according to input *value*
$\exists$	dependent product type	Σ	record fields depend on each other's *values*
natural deduction	type system		ensuring only "meaningful" programs
hypothesis	free variables		global variables, closures
implication elimination	function application		executing methods on arguments
implication introduction	λ-abstraction		parameters acting as local variables to method definitions

A Relationship Between Mathematics and Computing Science

Proposition P **is either** $true$ **or** $false$**.**

The Curry-Howard Correspondence —"Propositions as Types"

Logic	**Programming**		Example Use in Programming
$true$	singleton type		return type of side-effect only methods
$false$	empty type		return type for non-terminating methods
$\Rightarrow$	function type	$\rightarrow$	methods with an input and output type
$\wedge$	product type	$\times$	simple records of data and methods
$\vee$	sum type	$+$	enumerations or tagged unions
$\forall$	dependent function type	Π	return type varies according to input *value*
$\exists$	dependent product type	Σ	record fields depend on each other's *values*
natural deduction	type system		ensuring only "meaningful" programs
hypothesis	free variables		global variables, closures
implication elimination	function application		executing methods on arguments
implication introduction	λ-abstraction		parameters acting as local variables to method definitions

Type P **may have more than 2 elements!**

A Relationship Between Mathematics and Computing Science

Proposition *P* **is either** *true* **or** *false*.

The Curry-Howard Correspondence —"Propositions as Types"

Not An Isomorphism!

Logic	Programming		Example Use in Programming
true	singleton type		return type of side-effect only methods
false	empty type		return type for non-terminating methods
$\Rightarrow$	function type	$\to$	methods with an input and output type
$\wedge$	product type	$\times$	simple records of data and methods
$\vee$	sum type	$+$	enumerations or tagged unions
$\forall$	dependent function type	Π	return type varies according to input *value*
$\exists$	dependent product type	Σ	record fields depend on each other's *values*
natural deduction	type system		ensuring only "meaningful" programs
hypothesis	free variables		global variables, closures
implication elimination	function application		executing methods on arguments
implication introduction	λ-abstraction		parameters acting as local variables to method definitions

Type *P* **may have more than 2 elements!**

Adjunctions, à la `Prop`

1. Try to understand *complex notion* via a transformation to a *simpler notion.*

Adjunctions, à la `Prop`

1. Try to understand *complex notion* via a transformation to a *simpler notion*.
2. Methodology: Transport *complex*-problems into *simple*-setting where they are easier to solve, then go back.

Adjunctions, à la `Prop`

1. Try to understand *complex notion* via a transformation to a *simpler notion*.
2. Methodology: Transport *complex*-problems into *simple*-setting where they are easier to solve, then go back.
3. Usually transformation is not invertible, but has a best approximate inverse:

Adjunctions, à la `Prop`

1. Try to understand *complex notion* via a transformation to a *simpler notion.*
2. Methodology: Transport *complex*-problems into *simple*-setting where they are easier to solve, then go back.
3. Usually transformation is not invertible, but has a best approximate inverse:

Galois Connections

$$f \dashv g \qquad \equiv \qquad \forall x, y \bullet\ f\,x \leq_1 y \ \equiv\ x \leq_2 g\,y$$

Adjunctions, à la Prop

1. Try to understand *complex notion* via a transformation to a *simpler notion.*
2. Methodology: Transport *complex*-problems into *simple*-setting where they are easier to solve, then go back.
3. Usually transformation is not invertible, but has a best approximate inverse:

Galois Connections

$$f \dashv g \qquad \equiv \qquad \forall x, y \bullet\ f\,x \leq_1 y \;\equiv\; x \leq_2 g\,y$$

For example, we can understand the complicated notion of supremum by the trivial notion of constant function:

$$\forall h, z \;\bullet\; \sup\, h \leq z \;\equiv\; h \mathrel{\dot{\leq}} \mathsf{K}\, z$$

Easy now to show that supremum distributes over maximum
$$\sup(h \mathbin{\dot\sqcup} k) = \sup\ h \sqcup \sup k$$

1. Try to understand *complex notion* via a transformation to a *simpler notion*.
2. Methodology: Transport *complex*-problems into *simple*-setting where they are easier to solve, then go back.
3. Usually transformation is not invertible, but has a best approximate inverse:

Galois Connections

$$f \dashv g \qquad \equiv \qquad \forall x, y \bullet\ f\, x \leq_1 y \ \equiv\ x \leq_2 g\, y$$

For example, we can understand the complicated notion of supremum by the trivial notion of constant function:

$$\forall h, z \ \bullet\ \sup\ h \leq z \ \equiv\ h \mathrel{\dot\leq} \mathsf{K}\, z$$

Adjunctions, à la Prop

Easy now to show that supremum distributes over maximum
$\sup(h \mathbin{\dot{\sqcup}} k) = \sup h \sqcup \sup k$

1. Try to understand *complex notion* via a transformation to a *simpler notion.*
2. Methodology: Transport *complex*-problems into *simple*-setting where they are easier to solve, then go back.
3. Usually transformation is not invertible, but has a best approximate inverse:

Galois Connections

$$f \dashv g \qquad \equiv \qquad \forall x, y \bullet f\,x \leq_1 y \;\equiv\; x \leq_2 g\,y$$

For example, we can understand the complicated notion of supremum by the trivial notion of constant function:

$$\forall h, z \bullet \sup h \leq z \;\equiv\; h \mathrel{\dot{\leq}} \mathsf{K}\, z$$

Ubiquitous! Integer division: $(\times k) \dashv (\div n)$, floor: $\iota_{\mathbb{R}} \dashv \lfloor - \rfloor$,
Prefix extraction: $\mathsf{length} \times \mathsf{Id} \dashv \mathsf{take}$, Binary joins: $\max \dashv \Delta$, Power sets: $\bigcup \vdash \mathbb{P}$,
Unit & empty types, residuals, colimits, Kan extensions, Free $\vdash$ Forgetful

Adjunctions, Constructively

Proof relevance: We care about *which proof* witnesses the Galois Connection.

Adjunctions, Constructively

Proof relevance: We care about *which proof* witnesses the Galois Connection.
We make a few replacements,

Adjunctions, Constructively

Proof relevance: We care about *which proof* witnesses the Galois Connection.

We make a few replacements,

Order ‘$\leq$’	$\mapsto$	Morphism type constructor ‘$\longrightarrow$’

Adjunctions, Constructively

Proof relevance: We care about *which proof* witnesses the Galois Connection.
We make a few replacements,

Order ‘$\leq$’	$\mapsto$	Morphism type constructor ‘$\longrightarrow$’
Equivalence ‘$\equiv$’	$\mapsto$	Isomorphism ‘$\cong$’

Adjunctions, Constructively

Proof relevance: We care about *which proof* witnesses the Galois Connection.
We make a few replacements,

Order '$\leq$'	$\mapsto$	Morphism type constructor '$\longrightarrow$'
Equivalence '$\equiv$'	$\mapsto$	Isomorphism '$\cong$'
Functions f, g between sets	$\mapsto$	Functors F, G between categories

Adjunctions, Constructively

Proof relevance: We care about *which proof* witnesses the Galois Connection.
We make a few replacements,

Order '$\leq$'	$\mapsto$	Morphism type constructor '$\longrightarrow$'
Equivalence '$\equiv$'	$\mapsto$	Isomorphism '$\cong$'
Functions f, g between sets	$\mapsto$	Functors F, G between categories

Adjunctions, Global Characterisation

$$F \dashv G \qquad \equiv \qquad \forall x, y \bullet (F\,x \longrightarrow_1 y) \cong (x \longrightarrow_2 G\,y)$$

Where the isomorphism is natural in x and y.

Adjunctions, Constructively

Proof relevance: We care about *which proof* witnesses the Galois Connection.
We make a few replacements,

Order '$\leq$'	$\mapsto$	Morphism type constructor '$\longrightarrow$'
Equivalence '$\equiv$'	$\mapsto$	Isomorphism '$\cong$'
Functions f, g between sets	$\mapsto$	Functors F, G between categories

Adjunctions, Global Characterisation

$$F \dashv G \qquad \equiv \qquad \forall x, y \bullet (F\,x \longrightarrow_1 y) \cong (x \longrightarrow_2 G\,y)$$

Where the isomorphism is natural in x and y.

Adjunctions, Local Characterisation

$$\epsilon : FG \overset{\cdot}{\longrightarrow} \mathsf{Id} \qquad \text{and} \qquad \eta : \mathsf{Id} \overset{\cdot}{\longrightarrow} GF$$

such that the nonsensical "inverse equation $\mathsf{Id} = \epsilon \circ \eta$" holds.

η is called the unit since FG is a monad; dually ϵ is the counit since GF is a comonad.

Adjunctions, Constructively

Proof relevance: We care about *which proof* witnesses the Galois Connection.
We make a few replacements,

Order '$\leq$'	$\mapsto$	Morphism type constructor '$\longrightarrow$'
Equivalence '$\equiv$'	$\mapsto$	Isomorphism '$\cong$'
Functions f, g between sets	$\mapsto$	Functors F, G between categories

Adjunctions, Global Characterisation

$$F \dashv G \qquad \equiv \qquad \forall x, y \bullet (F\,x \longrightarrow_1 y) \cong (x \longrightarrow_2 G\,y)$$

Where the isomorphism is natural in x and y.

Adjunctions, Local Characterisation

$$\epsilon : FG \overset{.}{\longrightarrow} \mathsf{Id} \qquad \text{and} \qquad \eta : \mathsf{Id} \overset{.}{\longrightarrow} GF$$

such that the nonsensical "inverse equation $\mathsf{Id} = \epsilon \circ \eta$" holds.

η is called the unit since FG is a monad; dually ϵ is the counit since GF is a comonad.

η embeds, or *inserts*, "elements" *as* "singleton structures";
whereas ϵ *extracts* "elements" *from* "singleton structures."

Computing Science $\dashv$ Mathematics

Given an arbitrary type A,

Computing Science $\dashv$ Mathematics

Given an arbitrary type A,

Theory	**Structure**	Over	Equality
Carrier	Identity A	Type	Propositional
Pointed	Maybe A	Type	Propositional

Given an arbitrary type A,

Theory	**Structure**	Over	Equality
Carrier	Identity A	Type	Propositional
Pointed	Maybe A	Type	Propositional
Unary	$\mathbb{N} \times$ A	Type	Propositional
Involutive	A $\uplus$ A	Type	Propositional

Computing Science $\dashv$ Mathematics

Given an arbitrary type A,

Theory	**Structure**	Over	Equality
Carrier	Identity A	Type	Propositional
Pointed	Maybe A	Type	Propositional
Unary	$\mathbb{N}$ × A	Type	Propositional
Involutive	A ⊎ A	Type	Propositional
Magma	Tree A	Type	Propositional
Semigroup	NEList A	Type	Propositional

Computing Science $\dashv$ Mathematics

Given an arbitrary type A,

Theory	**Structure**	Over	Equality
Carrier	Identity A	Type	Propositional
Pointed	Maybe A	Type	Propositional
Unary	$\mathbb{N}$ × A	Type	Propositional
Involutive	A ⊎ A	Type	Propositional
Magma	Tree A	Type	Propositional
Semigroup	NEList A	Type	Propositional
Monoid	List A	Type	Propositional
Left Unital Semigroup	List A × $\mathbb{N}$	Type	Propositional
Right Unital Semigroup	$\mathbb{N}$ × List A	Type	Propositional

Computing Science ⊣ Mathematics

Given an arbitrary type A,

Theory	**Structure**	Over	Equality
Carrier	Identity A	Type	Propositional
Pointed	Maybe A	Type	Propositional
Unary	$\mathbb{N}$ × A	Type	Propositional
Involutive	A ⊎ A	Type	Propositional
Magma	Tree A	Type	Propositional
Semigroup	NEList A	Type	Propositional
Monoid	List A	Type	Propositional
Left Unital Semigroup	List A × $\mathbb{N}$	Type	Propositional
Right Unital Semigroup	$\mathbb{N}$ × List A	Type	Propositional
Commutative Monoid	Bag	Setoid	Proof-relevant permutations
Group	?	?	?
Abelian Group	Hybrid Sets	Setoid	Proof-relevant permutations
Idemp. Comm. Monoid	Set	Setoid	Logical equivalence

Computing Science $\dashv$ Mathematics

Given an arbitrary type A,

Theory	**Structure**	Over	Equality
Carrier	Identity A	Type	Propositional
Pointed	Maybe A	Type	Propositional
Unary	$\mathbb{N} \times$ A	Type	Propositional
Involutive	A $\uplus$ A	Type	Propositional
Magma	Tree A	Type	Propositional
Semigroup	NEList A	Type	Propositional
Monoid	List A	Type	Propositional
Left Unital Semigroup	List A $\times \mathbb{N}$	Type	Propositional
Right Unital Semigroup	$\mathbb{N} \times$ List A	Type	Propositional
Commutative Monoid	Bag	Setoid	Proof-relevant permutations
Group	?	?	?
Abelian Group	Hybrid Sets	Setoid	Proof-relevant permutations
Idemp. Comm. Monoid	Set	Setoid	Logical equivalence

$\diamond$ Free Structure is "the" **term language in normal form** associated to the theory.

Computing Science $\dashv$ Mathematics

Given an arbitrary type A,

Theory	**Structure**	Over	Equality
Carrier	Identity A	Type	Propositional
Pointed	Maybe A	Type	Propositional
Unary	$\mathbb{N} \times$ A	Type	Propositional
Involutive	A $\uplus$ A	Type	Propositional
Magma	Tree A	Type	Propositional
Semigroup	NEList A	Type	Propositional
Monoid	List A	Type	Propositional
Left Unital Semigroup	List A $\times \mathbb{N}$	Type	Propositional
Right Unital Semigroup	$\mathbb{N} \times$ List A	Type	Propositional
Commutative Monoid	Bag	Setoid	Proof-relevant permutations
Group	?	?	?
Abelian Group	Hybrid Sets	Setoid	Proof-relevant permutations
Idemp. Comm. Monoid	Set	Setoid	Logical equivalence

◇ Free Structure is "the" **term language in normal form** associated to the theory.

◇ $\bot$, $\top$, $\mathbb{B}$, $\mathbb{N}$, $\mathbb{Z}$ show up as *initial objects*.

Computing Science ⊣ Mathematics

Easy! ~Three Months!

Given an arbitrary type A,

Theory	**Structure**	Over	Equality
Carrier	Identity A	Type	Propositional
Pointed	Maybe A	Type	Propositional
Unary	$\mathbb{N} \times$ A	Type	Propositional
Involutive	A $\uplus$ A	Type	Propositional
Magma	Tree A	Type	Propositional
Semigroup	NEList A	Type	Propositional
Monoid	List A	Type	Propositional
Left Unital Semigroup	List A $\times\ \mathbb{N}$	Type	Propositional
Right Unital Semigroup	$\mathbb{N} \times$ List A	Type	Propositional
Commutative Monoid	Bag	Setoid	Proof-relevant permutations
Group	?	?	?
Abelian Group	Hybrid Sets	Setoid	Proof-relevant permutations
Idemp. Comm. Monoid	Set	Setoid	Logical equivalence

◇ Free Structure is “the” **term language in normal form** associated to the theory.

◇ $\bot$, $\top$, $\mathbb{B}$, $\mathbb{N}$, $\mathbb{Z}$ show up as *initial objects*.

Computing Science ⊣ Mathematics

Given an arbitrary type A,

Theory	**Structure**	Over	Equality
Carrier	Identity A	Type	Propositional
Pointed	Maybe A	Type	Propositional
Unary	$\mathbb{N} \times$ A	Type	Propositional
Involutive	A $\uplus$ A	Type	Propositional
Magma	Tree A	Type	Propositional
Semigroup	NEList A	Type	Propositional
Monoid	List A	Type	Propositional
Left Unital Semigroup	List A $\times \mathbb{N}$	Type	Propositional
Right Unital Semigroup	$\mathbb{N} \times$ List A	Type	Propositional
Commutative Monoid $\Leftarrow$	Bag	Setoid	Proof-relevant permutations
Group	?	?	?
Abelian Group	Hybrid Sets	Setoid	Proof-relevant permutations
Idemp. Comm. Monoid	Set	Setoid	Logical equivalence

Easy! ~Three Months!

This took one year!

◇ Free Structure is “the” **term language in normal form** associated to the theory.

◇ $\bot$, $\top$, $\mathbb{B}$, $\mathbb{N}$, $\mathbb{Z}$ show up as *initial objects*.

Given an arbitrary type A,

Theory	**Structure**	Over	Equality
Carrier	Identity A	Type	Propositional
Pointed	Maybe A	Type	Propositional
Unary	$\mathbb{N} \times$ A	Type	Propositional
Involutive	A $\uplus$ A	Type	Propositional
Magma	Tree A	Type	Propositional
Semigroup	NEList A	Type	Propositional
Monoid	List A	Type	Propositional
Left Unital Semigroup	List A $\times \mathbb{N}$	Type	Propositional
Right Unital Semigroup	$\mathbb{N} \times$ List A	Type	Propositional
Commutative Monoid $\Leftarrow$	Bag	Setoid	Proof-relevant permutations

Easy! ~Three Months!

This took one year!

Implementation issues!

Computing Science $\dashv$ Mathematics

Given an arbitrary type A,

Easy! ~Three Months!

Theory	**Structure**	Over	Equality
Carrier	Identity A	Type	Propositional
Pointed	Maybe A	Type	Propositional
Unary	$\mathbb{N} \times$ A	Type	Propositional
Involutive	A $\uplus$ A	Type	Propositional
Magma	Tree A	Type	Propositional
Semigroup	NEList A	Type	Propositional
Monoid	List A	Type	Propositional
Left Unital Semigroup	List A $\times \mathbb{N}$	Type	Propositional
Right Unital Semigroup	$\mathbb{N} \times$ List A	Type	Propositional
Commutative Monoid $\Leftarrow$	Bag	Setoid	Proof-relevant permutations

This took one year!

Implementation issues!

- $\diamond$ Inductive type $\Rightarrow$ Ordering! Thus not free!

Computing Science $\dashv$ Mathematics

Given an arbitrary type A,

Theory	**Structure**	Over	Equality
Carrier	Identity A	Type	Propositional
Pointed	Maybe A	Type	Propositional
Unary	$\mathbb{N} \times$ A	Type	Propositional
Involutive	A $\uplus$ A	Type	Propositional
Magma	Tree A	Type	Propositional
Semigroup	NEList A	Type	Propositional
Monoid	List A	Type	Propositional
Left Unital Semigroup	List A $\times \mathbb{N}$	Type	Propositional
Right Unital Semigroup	$\mathbb{N} \times$ List A	Type	Propositional
Commutative Monoid	Bag	Setoid	Proof-relevant permutations

Easy! ~Three Months!

$\Longleftarrow$ ***This took one year!***

Implementation issues!

- $\diamond$ Inductive type $\Rightarrow$ Ordering! Thus not free!
- $\diamond$ $A \to \mathbb{N}$ $\Rightarrow$ No finite support

Easy! ~Three Months!

Given an arbitrary type A,

Theory	Structure	Over	Equality
Carrier	Identity A	Type	Propositional
Pointed	Maybe A	Type	Propositional
Unary	$\mathbb{N} \times$ A	Type	Propositional
Involutive	A $\uplus$ A	Type	Propositional
Magma	Tree A	Type	Propositional
Semigroup	NEList A	Type	Propositional
Monoid	List A	Type	Propositional
Left Unital Semigroup	List A $\times\ \mathbb{N}$	Type	Propositional
Right Unital Semigroup	$\mathbb{N} \times$ List A	Type	Propositional
Commutative Monoid $\Leftarrow$	Bag	Setoid	Proof-relevant permutations

This took one year!

Implementation issues!

- Inductive type $\Rightarrow$ Ordering! Thus not free!
- $A \to \mathbb{N}$ $\Rightarrow$ No finite support
  $\Rightarrow$ Finiteness hard to express constructively!

Computing Science $\dashv$ Mathematics

Given an arbitrary type A,

Theory	Structure	Over	Equality
Carrier	Identity A	Type	Propositional
Pointed	Maybe A	Type	Propositional
Unary	$\mathbb{N} \times$ A	Type	Propositional
Involutive	A $\uplus$ A	Type	Propositional
Magma	Tree A	Type	Propositional
Semigroup	NEList A	Type	Propositional
Monoid	List A	Type	Propositional
Left Unital Semigroup	List A $\times\ \mathbb{N}$	Type	Propositional
Right Unital Semigroup	$\mathbb{N} \times$ List A	Type	Propositional
Commutative Monoid $\Leftarrow$	Bag	Setoid	Proof-relevant permutations

Easy! ~Three Months!

This took one year!

Implementation issues!

- ⋄ Inductive type $\Rightarrow$ Ordering! Thus not free!
- ⋄ $A \to \mathbb{N}$ $\Rightarrow$ No finite support
  - $\Rightarrow$ Finiteness hard to express constructively!
  - $\Rightarrow$ Decidable equality on A! Thus not free!

Easy! ~Three Months!

Given an arbitrary type A,

Theory	**Structure**	Over	Equality
Carrier	Identity A	Type	Propositional
Pointed	Maybe A	Type	Propositional
Unary	$\mathbb{N} \times$ A	Type	Propositional
Involutive	A $\uplus$ A	Type	Propositional
Magma	Tree A	Type	Propositional
Semigroup	NEList A	Type	Propositional
Monoid	List A	Type	Propositional
Left Unital Semigroup	List A $\times \mathbb{N}$	Type	Propositional
Right Unital Semigroup	$\mathbb{N} \times$ List A	Type	Propositional
Commutative Monoid $\Leftarrow$	Bag	Setoid	Proof-relevant permutations

This took one year!

Implementation issues!

- ⋄ Inductive type $\Rightarrow$ Ordering! Thus not free!
- ⋄ $A \to \mathbb{N}$ $\Rightarrow$ No finite support
  - $\Rightarrow$ Finiteness hard to express constructively!
  - $\Rightarrow$ Decidable equality on A! Thus not free!
- ⋄ List A with permutations! Works!

Computing Science $\dashv$ Mathematics

Easy: Three Months!

Given an arbitrary type A,

Theory	**Structure**	Over	Equality
Carrier	Identity A	Type	Propositional
Pointed	Maybe A	Type	Propositional
U[illegible]	[illegible]	[illegible]	[illegible]
In[illegible]	[illegible]	[illegible]	[illegible]
M[illegible]	[illegible]	[illegible]	[illegible]
Semigroup	NEList A	Type	Propositional
Monoid	List A	Type	Propositional
Left Unital Semigroup	List A $\times$ $\mathbb{N}$	Type	Propositional
Right Unital Semigroup	$\mathbb{N}$ $\times$ List A	Type	Propositional
Commutative Monoid $\Leftarrow$	Bag	Setoid	Proof-relevant permutations

Theorem :: Within Martin-Löf Type Theory, there's no *free* functor from *Types* to *Commutative Monoids* using propositional equality.

This took one year!

Implementation issues!

- $\diamond$ Inductive type $\Rightarrow$ Ordering! Thus not free!
- $\diamond$ $A \to \mathbb{N}$ $\Rightarrow$ No finite support
  - $\Rightarrow$ Finiteness hard to express constructively!
  - $\Rightarrow$ Decidable equality on A! Thus not free!
- $\diamond$ List A with permutations! Works!

Why the Mechanisation

$\sim$ 1000 lines of Agda code

Why the Mechanisation

$\sim$ 1000 lines of Agda code

Benefits of the formal approach

- Give solid grounding to folklore; e.g., there are no associative extensions of non-associative operators: There is no free functor from magmas to semigroups.

Why the Mechanisation

$\sim$ 1000 lines of Agda code

Benefits of the formal approach

- Give solid grounding to folklore; e.g., there are no associative extensions of non-associative operators: There is no free functor from magmas to semigroups.
- Unicode and mixfix operators $\Rightarrow$ Mechanisation is readable and not significantly more effort than a conventional presentation in LaTeX.

Why the Mechanisation

$\sim$ 1000 lines of Agda code

Benefits of the formal approach

- Give solid grounding to folklore; e.g., there are no associative extensions of non-associative operators: There is no free functor from magmas to semigroups.
- Unicode and mixfix operators $\Rightarrow$ Mechanisation is readable and not significantly more effort than a conventional presentation in LaTeX.
- Agda as a type checker for doing mathematics —manipulating symbols according to specified rules. Mechanised mathematical notation $\Rightarrow$ Confidence in results! Dispell silly conjectures/errors!

Why the Mechanisation

$\sim$ 1000 lines of Agda code

Benefits of the formal approach

- Give solid grounding to folklore; e.g., there are no associative extensions of non-associative operators: There is no free functor from magmas to semigroups.
- Unicode and mixfix operators $\Rightarrow$ Mechanisation is readable and not significantly more effort than a conventional presentation in LaTeX.
- Agda as a type checker for doing mathematics —manipulating symbols according to specified rules. Mechanised mathematical notation $\Rightarrow$ Confidence in results! Dispell silly conjectures/errors!
- Agda enables a natural treatment of theories and their direct use as modules of executable programs.

Why the Mechanisation

$\sim$ 1000 lines of Agda code

Benefits of the formal approach

- Give solid grounding to folklore; e.g., there are no associative extensions of non-associative operators: There is no free functor from magmas to semigroups.
- Unicode and mixfix operators $\Rightarrow$ Mechanisation is readable and not significantly more effort than a conventional presentation in LaTeX.
- Agda as a type checker for doing mathematics —manipulating symbols according to specified rules. Mechanised mathematical notation $\Rightarrow$ Confidence in results! Dispell silly conjectures/errors!
- Agda enables a natural treatment of theories and their direct use as modules of executable programs.
- Finally, formal proofs are fool-proof! **No "an exercise to the reader"!**

Why the Mechanisation

$\sim$ 1000 lines of Agda code

Benefits of the formal approach

- Give solid grounding to folklore; e.g., there are no associative extensions of non-associative operators: There is no free functor from magmas to semigroups.
- Unicode and mixfix operators $\Rightarrow$ Mechanisation is readable and not significantly more effort than a conventional presentation in LaTeX.
- Agda as a type checker for doing mathematics —manipulating symbols according to specified rules. Mechanised mathematical notation $\Rightarrow$ Confidence in results! Dispell silly conjectures/errors!
- Agda enables a natural treatment of theories and their direct use as modules of executable programs.
- Finally, formal proofs are fool-proof! **No "an exercise to the reader"!**

By working out the details we discovered

- Common recursion principles ; "Evaluate" the syntax under a given semantics.

Why the Mechanisation

$\sim$ 1000 lines of Agda code

Benefits of the formal approach

- Give solid grounding to folklore; e.g., there are no associative extensions of non-associative operators: There is no free functor from magmas to semigroups.
- Unicode and mixfix operators $\Rightarrow$ Mechanisation is readable and not significantly more effort than a conventional presentation in LaTeX.
- Agda as a type checker for doing mathematics —manipulating symbols according to specified rules. Mechanised mathematical notation $\Rightarrow$ Confidence in results! Dispell silly conjectures/errors!
- Agda enables a natural treatment of theories and their direct use as modules of executable programs.
- Finally, formal proofs are fool-proof! **No "an exercise to the reader"!**

By working out the details we discovered

- Common recursion principles ; "Evaluate" the syntax under a given semantics.
- `map` operators and their optimisation rewrites: $\mathsf{map}\,\mathsf{Id} \mapsto \mathsf{Id}$; $\mathsf{map}\, f \circ \mathsf{map}\, g \mapsto \mathsf{map}\,(f \circ g)$

Why the Mechanisation

$\sim$ 1000 lines of Agda code

Benefits of the formal approach

- Give solid grounding to folklore; e.g., there are no associative extensions of non-associative operators: There is no free functor from magmas to semigroups.
- Unicode and mixfix operators $\Rightarrow$ Mechanisation is readable and not significantly more effort than a conventional presentation in LaTeX.
- Agda as a type checker for doing mathematics —manipulating symbols according to specified rules. Mechanised mathematical notation $\Rightarrow$ Confidence in results! Dispell silly conjectures/errors!
- Agda enables a natural treatment of theories and their direct use as modules of executable programs.
- Finally, formal proofs are fool-proof! **No "an exercise to the reader"!**

By working out the details we discovered

- Common recursion principles ; "Evaluate" the syntax under a given semantics.
- `map` operators and their optimisation rewrites: $\mathsf{map}\,\mathsf{Id} \mapsto \mathsf{Id}$; $\mathsf{map}\, f \circ \mathsf{map}\, g \mapsto \mathsf{map}\,(f \circ g)$
- (Haskell's `return`) Unit transformations take on familiar forms: `[-]`, `Leaf`, `Just`, ... Along with proven optimisation laws: $\mathsf{map}\, f (\mathsf{return}\, x) \mapsto \mathsf{return}\,(f\, x)$

Why the Mechanisation

$\sim$ 1000 lines of Agda code

Benefits of the formal approach

- Give solid grounding to folklore; e.g., there are no associative extensions of non-associative operators: There is no free functor from magmas to semigroups.
- Unicode and mixfix operators $\Rightarrow$ Mechanisation is readable and not significantly more effort than a conventional presentation in LaTeX.
- Agda as a type checker for doing mathematics —manipulating symbols according to specified rules. Mechanised mathematical notation $\Rightarrow$ Confidence in results! Dispell silly conjectures/errors!
- Agda enables a natural treatment of theories and their direct use as modules of executable programs.
- Finally, formal proofs are fool-proof! **No "an exercise to the reader"!**

By working out the details we discovered

- Common recursion principles ; "Evaluate" the syntax under a given semantics.
- `map` operators and their optimisation rewrites: $\mathsf{map}\,\mathsf{Id} \mapsto \mathsf{Id}$; $\mathsf{map}\,f \circ \mathsf{map}\,g \mapsto \mathsf{map}\,(f \circ g)$
- (Haskell's `return`) Unit transformations take on familiar forms: `[-]`, `Leaf`, `Just`, ... Along with proven optimisation laws: $\mathsf{map}\,f(\mathsf{return}\,x) \mapsto \mathsf{return}\,(f\,x)$
- Boom Hierarchy does not necessarily work for type theory!

Why the Mechanisation

$\sim$ 1000 lines of Agda code

Benefits of the formal approach

- Give solid grounding to folklore; e.g., there are no associative extensions of non-associative operators: There is no free functor from magmas to semigroups.
- Unicode and mixfix operators $\Rightarrow$ Mechanisation is readable and not significantly more effort than a conventional presentation in LaTeX.
- Agda as a type checker for doing mathematics —manipulating symbols according to specified rules. Mechanised mathematical notation $\Rightarrow$ Confidence in results! Dispell silly conjectures/errors!
- Agda enables a natural treatment of theories and their direct use as modules of executable programs.
- Finally, formal proofs are fool-proof! **No "an exercise to the reader"!**

By working out the details we discovered

- Common recursion principles ; "Evaluate" the syntax under a given semantics.
- `map` operators and their optimisation rewrites: $\mathsf{map}\,\mathsf{Id} \mapsto \mathsf{Id}$; $\mathsf{map}\, f \circ \mathsf{map}\, g \mapsto \mathsf{map}\,(f \circ g)$
- (Haskell's `return`) Unit transformations take on familiar forms: `[-]`, `Leaf`, `Just`, ... Along with proven optimisation laws: $\mathsf{map}\, f (\mathsf{return}\, x) \mapsto \mathsf{return}\,(f\, x)$
- Boom Hierarchy does not necessarily work for type theory!
- New-ish relationships: Lists may be specified as a *free monoid* but may also be specified as *initial pointed indexed unary algebra*. Single-sorted unary algebras also have a surprising relationship with temporal logic.

Adventurous Avenues To Explore —Where do these show up?

Adventurous Avenues To Explore —Where do these show up?

Math theories yielding potential data structures

left-zero monoid (Prolog's cut!?), pointed unary (natural numbers!?), idempotent unary, commutative magma, pointed magma, quasigroup, loop, semilattice, medial magma, left semimedial magma, left distributive magma, idempotent magma, zeropotent magma, left unary magma, Steiner magma, null semigroup, BCI algebra, BCK algebra, squag, sloop, Moufang quasigroup, loop, left shelf, shelf, rack, spindle, quandle, Kei, involutive semigroup, band, rectangular band, hemigroup, pseudo inverse algebra, ringoid, left near semiring, near semiring, semifield, semiring, semirng, pre-dioid, dioid, star semiring, idempotent dioid, ring, commutative ring, idempotent semiring, Stone algebra, Kleene lattice, Kleene algebra, Heyting algebra, Goedel algebra, ortho lattice, directoid, semiheap, idempotent semiheap, heap, meadow, wheel.

Adventurous Avenues To Explore —Where do these show up?

Math theories yielding potential data structures

left-zero monoid (Prolog's cut!?), pointed unary (natural numbers!?), idempotent unary, commutative magma, pointed magma, quasigroup, loop, semilattice, medial magma, left semimedial magma, left distributive magma, idempotent magma, zeropotent magma, left unary magma, Steiner magma, null semigroup, BCI algebra, BCK algebra, squag, sloop, Moufang quasigroup, loop, left shelf, shelf, rack, spindle, quandle, Kei, involutive semigroup, band, rectangular band, hemigroup, pseudo inverse algebra, ringoid, left near semiring, near semiring, semifield, semiring, semirng, pre-dioid, dioid, star semiring, idempotent dioid, ring, commutative ring, idempotent semiring, Stone algebra, Kleene lattice, Kleene algebra, Heyting algebra, Goedel algebra, ortho lattice, directoid, semiheap, idempotent semiheap, heap, meadow, wheel.

Structures possibly arising from mathematical theories

Difference list (Yoneda on Monoids!?), stack, queue, finite map, rose tree, digraph, multigraph, partitions, oriented cycles, colorings, tri-colorings, hedges, derangements, ballots, commutative parenthesizations, linear order, permutations, even permutations, chains, oriented sets, even sets, octopus, vertebrae, automata (pointed indexed algebras!?).