Dependent pattern matching and proof-relevant unification

Public PhD defence

Jesper Cockx

DistriNet – KU Leuven

26 June 2017

How to tell a computer to do what you want?

Public PhD defence

Jesper Cockx

DistriNet – KU Leuven

26 June 2017

How to tell a computer to do what you want?

How to tell a computer to do what you want?

- By kicking?

How to tell a computer to do what you want?

- By kicking? ✖

How to tell a computer to do what you want?

- By kicking? ✖
- By yelling?

How to tell a computer to do what you want?

- By kicking? ✖
- By yelling? ✖

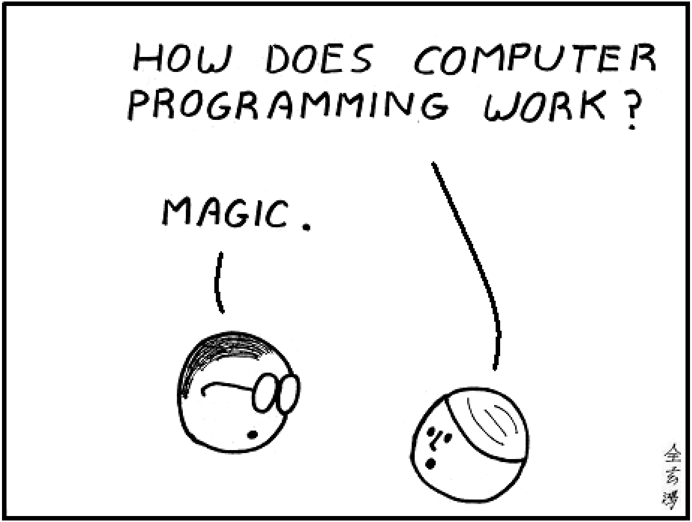
How to tell a computer to do what you want?

- By kicking? ✖
- By yelling? ✖
- By programming?

How to tell a computer to do what you want?

- By kicking? ✗
- By yelling? ✗
- By programming? ✓

What is programming?



What is programming?

Programming

$=$

What is programming?

Programming

=

telling the computer what to do

What is programming?

Programming

=

telling the computer what to do,
using a **programming language**.

What is programming?

Programming

=

telling the computer what to do,
using a **programming language**.

Examples: C, Java, JavaScript, Python, SQL, MatLab, Haskell, ML, ...

What is programming?

Programming

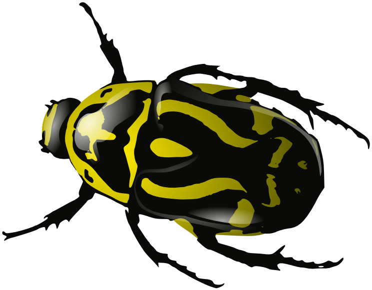
=

telling the computer what to do,
using a **programming language**.

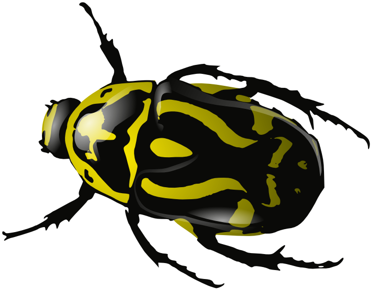
Examples: C, Java, JavaScript, Python, SQL, MatLab, Haskell, ML, **Agda**.

Programming is hard

Programming is hard



Programming is hard

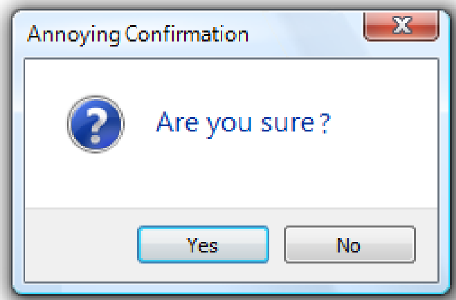


1. Why is programming hard?
2. How do type systems help?
3. What is pattern matching?
4. What is homotopy type theory?
5. What did I work on?

1. Why is programming hard?
2. How do type systems help?
3. What is pattern matching?
4. What is homotopy type theory?
5. What did I work on?

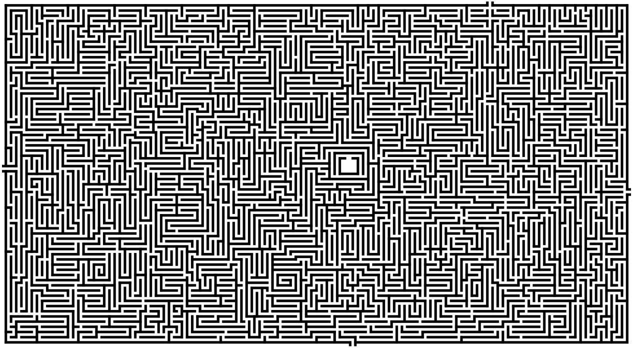
Programming is hard

- Computers take everything literally



Programming is hard

- Computers take everything literally
- The code has to cover all cases

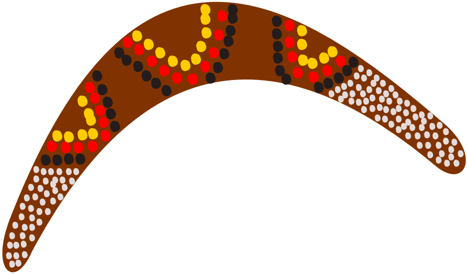


Programming is hard

- Computers take everything literally
- The code has to cover all cases
- Many pieces have to fit together

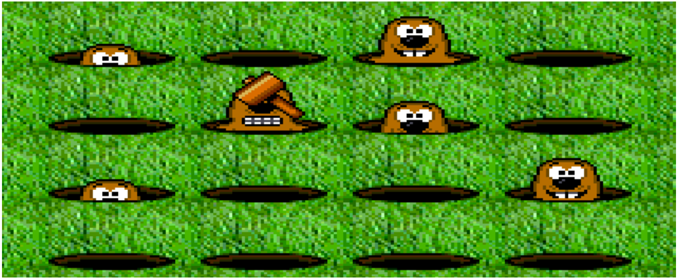
Programming is hard

- Computers take everything literally
- The code has to cover all cases
- Many pieces have to fit together
- You don't get immediate feedback



Programming is hard

- Computers take everything literally
- The code has to cover all cases
- Many pieces have to fit together
- You don't get immediate feedback
- Testing can't find all mistakes



1. Why is programming hard?
2. How do type systems help?
3. What is pattern matching?
4. What is homotopy type theory?
5. What did I work on?

What is a type system?

A **type system** *is a set of rules that assign a property called* **type** *to various constructs a computer program consists of.*

(paraphrased from Wikipedia)

What is a type system?

A **type system** *is a set of rules that assign a property called* **type** *to various constructs a computer program consists of.*

The main purpose of a type system is to reduce possibilities for bugs in computer programs.

(paraphrased from Wikipedia)

What is a type system?

The **term** a has **type** T

$$a : T$$

What is a type system?

The **term** a has **type** T

$$a : T$$

`donut` : `Pastry`

What is a type system?

The **term** a has **type** T

$$a : T$$

`donut` : `Pastry`

`dragon` : `Monster`

Simple type theory (1940)

Base types:

Pastry, Monster, ...

Alonzo
Church

Simple type theory (1940)

Base types:

$\texttt{Pastry}$, $\texttt{Monster}$, ...

Function types:

$$A \to B$$

Alonzo Church

Simple type theory (1940)

Base types:

$\texttt{Pastry}, \texttt{Monster}, \ldots$

Function types:

$$A \to B$$

Function application:

If $f : A \to B$ and $x : A$,
then $f\ x : B$

Alonzo Church

How do types help to write correct code?

$$\texttt{eat} : \texttt{Pastry} \to \texttt{IO Unit}$$

How do types help to write correct code?

$\mathtt{eat} : \mathtt{Pastry} \to \mathtt{IO\ Unit}$

$\vdots$

$\mathtt{eat\ donut}$

How do types help to write correct code?

$\texttt{eat} : \texttt{Pastry} \to \texttt{IO Unit}$

$\vdots$

$\texttt{eat donut}$ ✓

How do types help to write correct code?

$\texttt{eat} : \texttt{Pastry} \rightarrow \texttt{IO Unit}$

$\vdots$

`eat donut` ✓

`eat dragon`

How do types help to write correct code?

$$\texttt{eat} : \texttt{Pastry} \to \texttt{IO Unit}$$

$\vdots$

`eat donut` ✓

`eat dragon` ✗

Type error: A `Monster` is not a `Pastry`!

Dependent type theory (1972)

Per
Martin-Löf

A **dependent type** is a family of types, depending on a term of a **base type**.

Dependent type theory (1972)

Per
Martin-Löf

A **dependent type** is a family of types, depending on a term of a **base type**.

$$\texttt{Monster}$$
$$\Downarrow$$
$$\texttt{small Monster},$$
$$\texttt{large Monster},$$
$$\ldots$$

1. Why is programming hard?
2. How do type systems help?
3. What is pattern matching?
4. What is homotopy type theory?
5. What did I work on?

Declarative programming

Declarative programming

=

say **what** you want

Declarative programming

Declarative programming

=

say **what** you want,

not **how** to do it.

Declarative programming

That is the very purpose of declarative programming – to make it more likely that we mean what we say by improving our ability to say what we mean.

— Conor McBride (2003)

Pattern matching

Write programs by giving equations:

$$\texttt{flavour} : \texttt{Food} \to \texttt{Flavour}$$

Pattern matching

Write programs by giving equations:

$$
\begin{aligned}
&\texttt{flavour} : \texttt{Food} \to \texttt{Flavour} \\
&\texttt{flavour}\ \texttt{pizza} \quad = \texttt{cheesy}
\end{aligned}
$$

Pattern matching

Write programs by giving equations:

$$\begin{aligned}
\texttt{flavour} &: \texttt{Food} \to \texttt{Flavour} \\
\texttt{flavour}\ \texttt{pizza} &= \texttt{cheesy} \\
\texttt{flavour}\ \texttt{moelleux} &= \texttt{chocolaty}
\end{aligned}$$

The AGDA language

The AGDA language

A purely functional language

The AGDA language

A purely functional language

... for writing programs and proofs

The AGDA language

A purely functional language

... for writing programs and proofs

... with datatypes and pattern matching

The AGDA language

A purely functional language

... for writing programs and proofs

... with datatypes and pattern matching

... with first-class dependent types

The AGDA language

A purely functional language

. . . for writing programs and proofs

. . . with datatypes and pattern matching

. . . with first-class dependent types

. . . with support for interactive development

The Agda language

A purely functional language

… for writing programs and proofs

… with datatypes and pattern matching

… with first-class dependent types

… with support for interactive development

Demo time!

Dependent pattern matching (1992)

By pattern matching, we can learn something about the type.

Thierry Coquand

Dependent pattern matching (1992)

By pattern matching, we can learn something about the type.

```
ingredients pizza :
  List (cheesy Ingredient)
```

Thierry Coquand

Checking definitions by pattern matching

Checking definitions by pattern matching

Unification

=

Finding ways to make two terms equal

Checking definitions by pattern matching

Unification

$=$

Finding ways to make two terms equal, by solving equations step by step.

Specialization by unification: The solution rule

We can make x equal to `cheesy`

Specialization by unification: The solution rule

We can make x equal to `cheesy`,
by replacing x with `cheesy` everywhere.

Specialization by unification: The solution rule

We can make x equal to `cheesy`,
by replacing x with `cheesy` everywhere.

$$\texttt{ingredients} : \{x : \texttt{Flavour}\} \to x\ \texttt{Food} \to \texttt{List}\ (x\ \texttt{Ingredient})$$

Specialization by unification: The solution rule

We can make x equal to $\texttt{cheesy}$,
by replacing x with $\texttt{cheesy}$ everywhere.

$$\begin{gathered}\texttt{ingredients} : \{x : \texttt{Flavour}\} \to \\ x\ \texttt{Food} \to \texttt{List}\ (x\ \texttt{Ingredient}) \\ \Downarrow \\ \texttt{ingredients}\ \texttt{pizza} : \\ \texttt{List}\ (\texttt{cheesy}\ \texttt{Ingredient})\end{gathered}$$

Specialization by unification: The deletion rule

We can make `cheesy` equal to `cheesy`

Specialization by unification: The deletion rule

We can make `cheesy` equal to `cheesy`, by doing nothing.

Specialization by unification: The deletion rule

We can make `cheesy` equal to `cheesy`, by doing nothing.

$$\texttt{amount-of-cheese} : \\ \quad \texttt{cheesy Food} \to \texttt{Amount}$$

Specialization by unification: The deletion rule

We can make `cheesy` equal to `cheesy`, by doing nothing.

$$\texttt{amount-of-cheese} : \texttt{cheesy Food} \rightarrow \texttt{Amount}$$

$$\Downarrow$$

$$\texttt{amount-of-cheese pizza} : \texttt{Amount}$$

Specialization by unification: The conflict rule

`cheesy` can never be equal to `chocolaty`

Specialization by unification: The conflict rule

`cheesy` can never be equal to `chocolaty`, so we can safely skip the case.

Specialization by unification: The conflict rule

`cheesy` can never be equal to `chocolaty`, so we can safely skip the case.

```
amount-of-cheese :
      cheesy Food → Amount
```

Specialization by unification: The conflict rule

`cheesy` can never be equal to `chocolaty`, so we can safely skip the case.

$$\begin{array}{l}\texttt{amount-of-cheese} : \\ \qquad \texttt{cheesy}\ \texttt{Food} \to \texttt{Amount} \\ \qquad\qquad \not\Downarrow \end{array}$$

No case for `amount-of-cheese` `moelleux`!

1. Why is programming hard?

2. How do type systems help?

3. What is pattern matching?

4. What is homotopy type theory?

5. What did I work on?

Homotopy type theory

Homotopy type theory

$=$

a new type system with weirdly shaped types

Homotopy type theory

Homotopy type theory

=

a new type system with weirdly shaped types,
such as **donuts**

Homotopy type theory

Homotopy type theory

=

a new type system with weirdly shaped types,
such as **donuts** and **pancakes**.

The univalence axiom (2009)

Vladimir
Voevodsky

The univalence axiom (2009)

Vladimir Voevodsky

"Isomorphic types can be identified."

The univalence axiom (2009)

Vladimir
Voevodsky

"Isomorphic types
can be identified."

$$(A \equiv B) \simeq (A \simeq B)$$

The univalence axiom (2009)

`Flavour` is equal to `Bool` in two ways:

`Flavour`

`cheesy` `chocolaty`

The univalence axiom (2009)

`Flavour` is equal to `Bool` in two ways:

`Flavour`

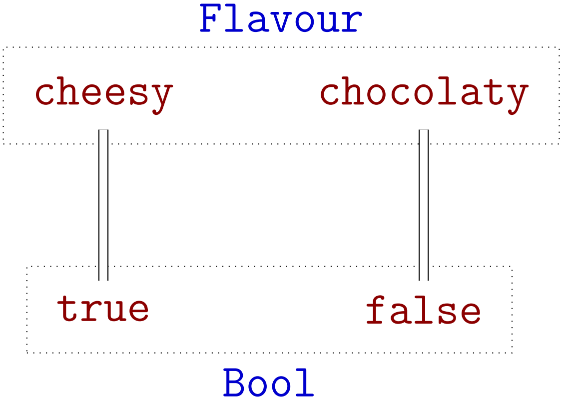
`cheesy`	`chocolaty`

`true`	`false`

`Bool`

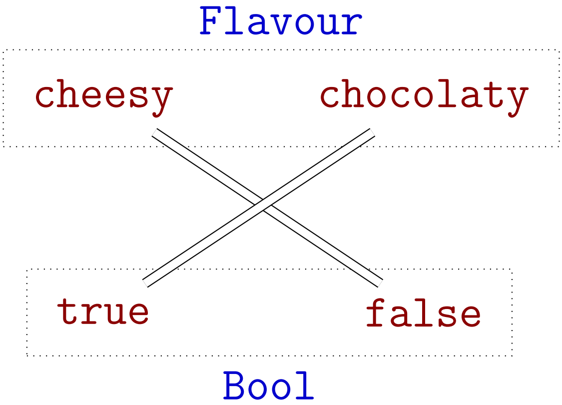
The univalence axiom (2009)

`Flavour` is equal to `Bool` in two ways:



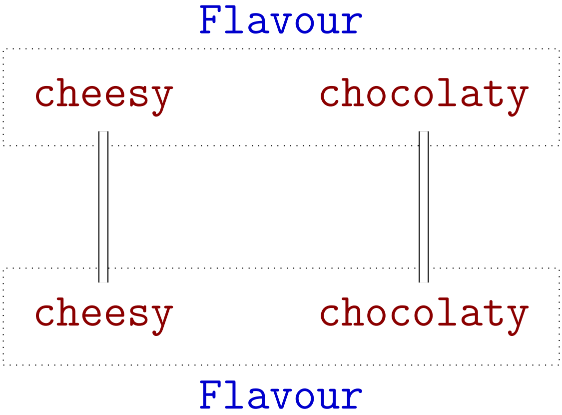
The univalence axiom (2009)

Flavour is equal to Bool in two ways:



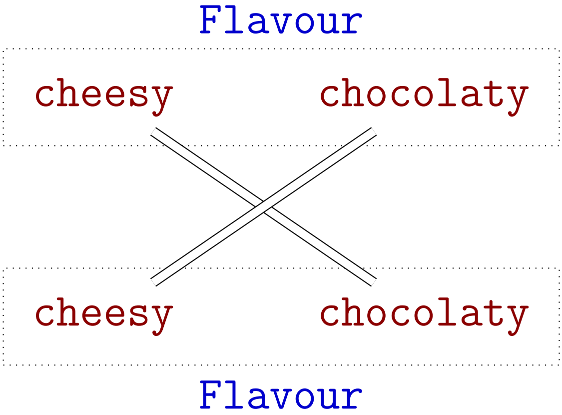
The univalence axiom (2009)

`Flavour` is equal to *itself* in two ways:



The univalence axiom (2009)

`Flavour` is equal to *itself* in two ways:



1. Why is programming hard?
2. How do type systems help?
3. What is pattern matching?
4. What is homotopy type theory?
5. What did I work on?

My process of working on Agda

1. Discover a new problem

My process of working on Agda

1. Discover a new problem
2. Search for the cause of the problem

My process of working on Agda

1. Discover a new problem
2. Search for the cause of the problem
3. Think of a solution

My process of working on Agda

1. Discover a new problem
2. Search for the cause of the problem
3. Think of a solution
4. Implement the solution

My process of working on Agda

1. Discover a new problem
2. Search for the cause of the problem
3. Think of a solution
4. Implement the solution
5. Prove that the solution works

My process of working on Agda

1. Discover a new problem
2. Search for the cause of the problem
3. Think of a solution
4. Implement the solution
5. Prove that the solution works
6. Write a paper about the solution

Pattern matching without K

Problem. Dependent pattern matching doesn't work in homotopy type theory.

Pattern matching without K

Problem. Dependent pattern matching doesn't work in homotopy type theory.

`Flavour` is equal to *itself* in two ways

Pattern matching without K

Problem. Dependent pattern matching doesn't work in homotopy type theory.

`Flavour` is equal to *itself* in two ways, so we cannot use the deletion rule!

Pattern matching without K

Problem. Dependent pattern matching doesn't work in homotopy type theory.

`Flavour` is equal to *itself* in two ways, so we cannot use the deletion rule!

My contribution. A new version of pattern matching that doesn't rely on deletion.

Proof-relevant unification

Problem. Unification doesn't consider the ways in which terms can be made equal.

Proof-relevant unification

Problem. Unification doesn't consider the ways in which terms can be made equal.

We call these 'ways to make terms equal' *equality proofs*.

Proof-relevant unification

Problem. Unification doesn't consider the ways in which terms can be made equal.

We call these 'ways to make terms equal' *equality proofs*.

My contribution. A unification algorithm that takes equality proofs into account.

Eliminating pattern matching

Problem. How can we be sure pattern matching doesn't cause any problems?

Eliminating pattern matching

Problem. How can we be sure pattern matching doesn't cause any problems?

In a standard type theory, we only have *datatype eliminators*.

Eliminating pattern matching

Problem. How can we be sure pattern matching doesn't cause any problems?

In a standard type theory, we only have *datatype eliminators*.

Main theorem. Any definition by pattern matching can be translated to eliminators.

Eliminating pattern matching

Problem. How can we be sure pattern matching doesn't cause any problems?

In a standard type theory, we only have *datatype eliminators*.

Main theorem. Any definition by pattern matching can be translated to eliminators.

Proof. See my thesis. $\square$

From pattern matching ...

$$
\begin{array}{l}
\mathtt{antisym} : (x\ y : \mathbb{N}) \to (x \leq y) \to (y \leq x) \to (x \equiv y) \\
\mathtt{antisym}\ .\mathtt{zero}\quad .\mathtt{zero}\qquad (\mathtt{lz}\ \lfloor \mathtt{zero} \rfloor)\ \ (\mathtt{lz}\ \lfloor \mathtt{zero} \rfloor) = \mathtt{refl} \\
\mathtt{antisym}\ .(\mathtt{suc}\ k)\ .(\mathtt{suc}\ l)\ \ (\mathtt{ls}\ k\ l\ u)\qquad (\mathtt{ls}\ .l\ .k\ v) \\
\qquad = \mathtt{cong}\ \mathtt{suc}\ (\mathtt{antisym}\ k\ l\ u\ v)
\end{array}
$$

... to eliminators.

$$
\begin{array}{l}
\mathtt{antisym} : (x\ y : \mathbb{N}) \to (x \leq y) \to (y \leq x) \to (x \equiv y) \\
\mathtt{antisym} = \mathtt{elim}_{\leq}\ (\lambda x\, y\, u.\, y \leq x \to x \equiv_{\mathbb{N}} y) \\
\quad (\lambda l\, v.\, \mathtt{elim}_{\leq}\ (\lambda y\, x\, v.\, x \equiv_{\mathbb{N}} \mathtt{zero} \to x \equiv_{\mathbb{N}} y) \\
\qquad (\lambda x\, e.\, e) \\
\qquad (\lambda l\, k\, \_\, \_\, e.\, \mathtt{elim}_{\bot}\ (\mathtt{suc}\ k \equiv_{\mathbb{N}} \mathtt{suc}\ l)\ (\mathtt{noConf}_{\mathbb{N}}\ (\mathtt{suc}\ k)\ \mathtt{zero}\ e)) \\
\qquad l\ \mathtt{zero}\ v\ \mathtt{refl}) \\
\quad (\lambda k\, l\, \_\, H\, v.\, \mathtt{cong}\ \mathtt{suc}\ (H \\
\qquad (\mathtt{elim}_{\leq}\ (\lambda x\, y\, \_.\, x \equiv_{\mathbb{N}} \mathtt{suc}\ l \to u \equiv_{\mathbb{N}} \mathtt{suc}\ k \to l \leq k) \\
\qquad\quad (\lambda l'\, e\, \_.\, \mathtt{elim}_{\bot}\ (l \leq k)\ (\mathtt{noConf}_{\mathbb{N}}\ \mathtt{zero}\ (\mathtt{suc}\ l)\ e)) \\
\qquad\quad (\lambda k'\, l'\, v'\, \_\, e_1\, e_2.\, \mathtt{subst}\ (\lambda n.\ n \leq k) \\
\qquad\qquad (\mathtt{noConf}_{\mathbb{N}}\ (\mathtt{suc}\ k')\ (\mathtt{suc}\ l)\ e_1) \\
\qquad\qquad (\mathtt{subst}\ (\lambda m.\ k' \leq m)\ (\mathtt{noConf}_{\mathbb{N}}\ (\mathtt{suc}\ l')\ (\mathtt{suc}\ k)\ e_2)\ v')) \\
\qquad\quad (\mathtt{suc}\ l)\ (\mathtt{suc}\ k)\ v\ \mathtt{refl}\ \mathtt{refl})))
\end{array}
$$

Take-home message

A simple type system can stop you from trying to eat a dragon. . .

Take-home message

A simple type system can stop you
from trying to eat a dragon…

…but if you don't like chocolate on
your pizza, you need dependent types.