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? (X)
- 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?

How does computer programming work?

MAGIC.
What is programming?

Programming

=
What is programming?

Programming = telling the computer what to do

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

![Annoying Confirmation]

Are you sure?

Yes  No
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 $T$

$$a : T$$

$\text{donut} : \text{Pastry}$

$\text{dragon} : \text{Monster}$
Simple type theory (1940)

Base types:

Pastry, Monster, ...

Alonzo Church
Simple type theory (1940)

Base types:

Pastry, Monster, ...

Function types:

\[ A \to B \]
Simple type theory (1940)

Base types:

Pastry, Monster, ...

Function types:

\[ A \rightarrow B \]

Function application:

If \( f : A \rightarrow B \) and \( x : A \), then \( f \, x : B \)

Alonzo Church
How do types help to write correct code?

\[
\text{eat : Pastry} \rightarrow \text{IO Unit}
\]
How do types help to write correct code?

eat : Pastry → IO Unit

: 

eat donut
How do types help to write correct code?

eat : Pastry \rightarrow IO Unit

eat donut
How do types help to write correct code?

eat : Pastry \rightarrow IO Unit

: 

eat donut ✔

eat dragon
How do types help to write correct code?

\[
eat : \text{Pastry} \rightarrow \text{IO Unit}
\]

- eat donut 😊
- eat dragon 😞

Type error: A Monster is not a Pastry!
Dependent type theory (1972)

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

Per Martin-Löf
Dependent type theory (1972)

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

Monster

\[ \downarrow \]
small Monster,
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.

Pattern matching

Write programs by giving equations:

\[ \text{flavour} : \text{Food} \rightarrow \text{Flavour} \]
Pattern matching

Write programs by giving equations:

\[
\text{flavour} : \text{Food} \rightarrow \text{Flavour} \\
\text{flavour pizza} = \text{cheesy}
\]
Pattern matching

Write programs by giving equations:

\[
\text{flavour : Food } \rightarrow \text{ Flavour} \\
\text{flavour pizza } = \text{ cheesy} \\
\text{flavour moelleux } = \text{ chocolaty}
\]
The VJDA language
The *Veda* language

A purely functional language
A purely functional language
... for writing programs and proofs
The VGDA language

A purely functional language

... for writing programs and proofs

... with datatypes and pattern matching
The language

A purely functional language

... for writing programs and proofs
... with datatypes and pattern matching
... with first-class dependent types
The **V GTA** 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 VGLA 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!
By pattern matching, we can learn something about the type.
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

Uniﬁcation = Finding ways to make two terms equal, by solving equations step by step.
Checking definitions by pattern matching

Unification

\[ \overset{=}{\text{Finding ways to make two terms equal}} \]
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 \textcolor{red}{cheesy} everywhere.

\[
\text{ingredients} : \{x : \text{Flavour}\} \rightarrow \\
x \text{ Food} \rightarrow \text{List}(x \text{ Ingredient})
\]
Specialization by unification: The solution rule

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

\[
\text{ingredients}: \{x: \text{Flavour}\} \rightarrow \\
x \rightarrow \text{List}(x \text{ Ingredient})
\]

\[
\Downarrow
\]

\[
\text{ingredients pizza}: \\
\text{List}((\text{cheesy Ingredient})
\]
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.

amount-of-cheese :

cheesy Food → Amount
Specialization by unification: The deletion rule

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

\[
\text{amount-of-cheese}:
\]

\[
\text{cheesy Food} \rightarrow \text{Amount}
\]

\[
\downarrow
\]

\[
\text{amount-of-cheese pizza} : \text{Amount}
\]
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.

\[
\text{amount-of-cheese} : \\
\text{cheesy Food} \rightarrow \text{Amount}
\]
Specialization by unification: The conflict rule

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

\[
\text{amount-of-cheese :}
\]

\[
\text{cheesy Food } \rightarrow \text{ Amount}
\]

No case for \text{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

Isomorphic types can be identified.
The univalence axiom (2009)

“Isomorphic types can be identified.”

Vladimir Voevodsky
The univalence axiom (2009)

Isomorphic types can be identified.

\[(A \equiv B) \leadsto (A \simeq B)\]

Vladimir Voevodsky
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**

<table>
<thead>
<tr>
<th>cheesy</th>
<th>chocolaty</th>
</tr>
</thead>
</table>

**Bool**

| true       | false     |
The univalence axiom (2009)

Flavour is equal to \textbf{Bool} in two ways:

\begin{center}
\begin{tikzpicture}
    \node (flavour) at (0,0) {Flavour};
    \node (cheesy) at (-2,-2) {cheesy};
    \node (chocolaty) at (2,-2) {chocolaty};
    \node (true) at (-2,-4) {true};
    \node (false) at (2,-4) {false};

    \draw (cheesy) -- (true);
    \draw (chocolaty) -- (false);
\end{tikzpicture}
\end{center}
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 *itself* in two ways:

Flavour

<table>
<thead>
<tr>
<th>cheesy</th>
<th>chocolaty</th>
</tr>
</thead>
</table>

| cheesy | chocolaty |

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

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

\[
\text{antisym} : (x \ y : \mathbb{N}) \rightarrow (x \leq y) \rightarrow (y \leq x) \rightarrow (x \equiv y)
\]

\[
\text{antisym} \ .\text{zero} \ .\text{zero} \ (\text{lz} \lfloor \text{zero} \rfloor) \ (\text{lz} \lfloor \text{zero} \rfloor) = \text{refl}
\]

\[
\text{antisym} \ .(\text{suc} \ k) \ .(\text{suc} \ l) \ (\text{ls} \ k \ l \ u) \ (\text{ls} \ .\ l \ .\ k \ v) = \text{cong suc} \ (\text{antisym} \ k \ l \ u \ v)
\]
antisym: \((x y : \mathbb{N}) \rightarrow (x \leq y) \rightarrow (y \leq x) \rightarrow (x \equiv y)\)

\[
\text{antisym} = \text{elim}_{\leq} (\lambda x y u. y \leq x \rightarrow x \equiv_{\mathbb{N}} y) \\
(\lambda l v. \text{elim}_{\leq} (\lambda y x v. x \equiv_{\mathbb{N}} \text{zero} \rightarrow x \equiv_{\mathbb{N}} y) \\
(\lambda x e. e) \\
(\lambda l k \_e. \text{elim}_{\bot} (\text{suc} k \equiv_{\mathbb{N}} \text{suc} l) (\text{noConf}_{\mathbb{N}} (\text{suc} k) \text{zero} e)) \\
l \text{zero} v \text{refl}) \\
(\lambda k l H v. \text{cong suc} (H) \\
(\text{elim}_{\leq} (\lambda x y \_x \equiv_{\mathbb{N}} \text{suc} l \rightarrow u \equiv_{\mathbb{N}} \text{suc} k \rightarrow l \leq k) \\
(\lambda l' e \_e. \text{elim}_{\bot} (l \leq k) (\text{noConf}_{\mathbb{N}} \text{zero} (\text{suc} l) e)) \\
(\lambda k' l' v' \_e_1 e_2. \text{subst} (\lambda n. n \leq k) \\
(\text{noConf}_{\mathbb{N}} (\text{suc} k') (\text{suc} l) e_1) \\
(\text{subst} (\lambda m. k' \leq m) (\text{noConf}_{\mathbb{N}} (\text{suc} l') (\text{suc} k) e_2) v')) \\
(\text{suc} l) (\text{suc} k) v \text{refl refl}))
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.