Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Types, Universes and Everything

Andres Löh

Dept. of Information and Computing Sciences, Utrecht University
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
Web pages: `http://www.cs.uu.nl/wiki/Center`

May 26, 2010

This talk

- The importance of strong type systems for programming.
- Current research topic: dependently typed programming.

Example (C#)

stolen from Tim Sweeney's POPL 2006 talk

Given a vertex array and an index array, let us read the indexed vertices, transform them, and write the result into a new array.

Example (C#)
stolen from Tim Sweeney's POPL 2006 talk

Given a vertex array and an index array, let us read the indexed vertices, transform them, and write the result into a new array.

```
Vertex[] Transform (Vertex[] Vertices, int[] Indices, Matrix m)
{
  Vertex[] Result = new Vertex[Indices.length];
  for (int i = 0; i < Indices.length; i++)
    Result[i] = Transform (m, Vertices[Indices[i]]);
  return Result;
}
```

Example (C#)

stolen from Tim Sweeney's POPL 2006 talk

Given a vertex array and an index array, let us read the indexed vertices, transform them, and write the result into a new array.

```
Vertex[] Transform (Vertex[] Vertices, int[] Indices, Matrix m)
{
  Vertex[] Result = new Vertex[Indices.length];
  for (int i = 0; i < Indices.length; i++)
    Result[i] = Transform (m, Vertices[Indices[i]]);
  return Result;
}
```

Can anything go wrong?

Example (C#)

stolen from Tim Sweeney's POPL 2006 talk

Given a vertex array and an index array, let us read the indexed vertices, transform them, and write the result into a new array.

can be null (Vertex[]) can be null (int[]) can be null (Matrix)

```
Vertex[] Transform (Vertex[] Vertices, int[] Indices, Matrix m)
{
  Vertex[] Result = new Vertex[Indices.length];
  for (int i = 0; i < Indices.length; i++)
    Result[i] = Transform (m, Vertices[Indices[i]]);
  return Result;
}
```

Can anything go wrong?

Example (C#)

stolen from Tim Sweeney's POPL 2006 talk

Given a vertex array and an index array, let us read the indexed vertices, transform them, and write the result into a new array.

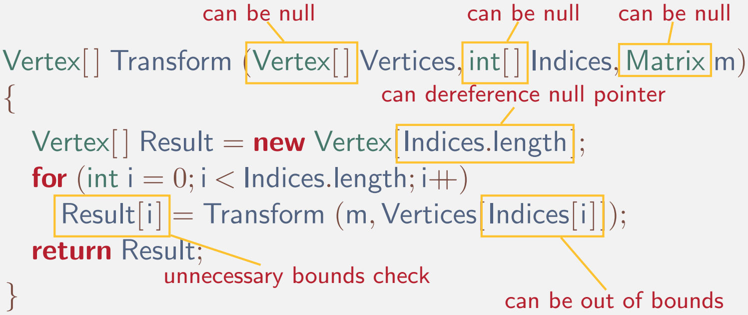
```
                     can be null             can be null   can be null
Vertex[] Transform (Vertex[] Vertices, int[] Indices, Matrix m)
{                                can dereference null pointer
  Vertex[] Result = new Vertex[Indices.length];
  for (int i = 0; i < Indices.length; i++)
    Result[i] = Transform (m, Vertices[Indices[i]]);
  return Result;
}
```

Can anything go wrong?

Example (C#)

stolen from Tim Sweeney's POPL 2006 talk

Given a vertex array and an index array, let us read the indexed vertices, transform them, and write the result into a new array.



```
Vertex[] Transform (Vertex[] Vertices, int[] Indices, Matrix m)
{
  Vertex[] Result = new Vertex[Indices.length];
  for (int i = 0; i < Indices.length; i++)
    Result[i] = Transform (m, Vertices[Indices[i]]);
  return Result;
}
```

can be null

can be null

can be null

can dereference null pointer

unnecessary bounds check

can be out of bounds

Can anything go wrong?

The problem

Types often cannot express the properties of programs sufficiently well.

Potential solutions

(Lots of) Testing

Potential solutions

(Lots of) Testing

Assertions and Contracts

Potential solutions

(Lots of) Testing

Assertions and Contracts

External verification

Potential solutions

(Lots of) Testing

Assertions and Contracts

External verification

. . .

The problem in the real world

The CWE/SANS Top 25 Most Dangerous Programming Errors

- Failure to Preserve Web Page Structure (“Cross-site Scripting”)
- Failure to Preserve SQL Query Structure (“SQL Injection”)
- Failure to Preserve OS Command Structure
- Buffer Copy without Checking Size of Input (“Buffer Overflow”)
- Improper Limitation of a Pathname to a Restricted Directory
- Improper Check for Unusual or Exceptional Conditions
- Improper Validation of Array Index
- Integer Overflow or Wraparound
- Missing Encryption of Sensitive Data
- . . .

Better types?

```
Vertex[] Transform (Vertex[] Vertices, int[] Indices, Matrix m)
{
  Vertex[] Result = new Vertex[Indices.length];
  for (int i = 0; i < Indices.length; i++)
    Result[i] = Transform (m, Vertices[Indices[i]]);
  return Result;
}
```

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Better types!

```
Transform {n : Nat}
          (Vertices : Vector Vertex n)
          (Indices : Buffer (m : Nat where m < n))
          (m : Matrix)
  : Vector Vertex (Indices.length) =
  [Transform (m, Vertices[i]) where i ← Indices]
```

Better types!

$$
\begin{aligned}
&\mathsf{Transform}\ \{\mathsf{n} : \mathsf{Nat}\}\\
&\qquad\qquad(\mathsf{Vertices} : \overset{\text{types do not admit null values}}{\boxed{\mathsf{Vector}}}\ \mathsf{Vertex}\ \mathsf{n})\\
&\qquad\qquad(\mathsf{Indices} : \mathsf{Buffer}\ (\mathsf{m} : \mathsf{Nat}\ \textbf{where}\ \mathsf{m} < \mathsf{n}))\\
&\qquad\qquad(\mathsf{m} : \mathsf{Matrix})\\
&\quad : \mathsf{Vector}\ \mathsf{Vertex}\ (\mathsf{Indices.length}) =\\
&\quad [\mathsf{Transform}\ (\mathsf{m}, \mathsf{Vertices}[\mathsf{i}])\ \textbf{where}\ \mathsf{i} \leftarrow \mathsf{Indices}]
\end{aligned}
$$

Better types!

quantification over a natural number

types do not admit null values

$$\begin{array}{l}
\mathrm{Transform}\ \{\boxed{\mathrm{n}} : \mathrm{Nat}\} \\
\qquad\qquad (\mathrm{Vertices} : \boxed{\mathrm{Vector}}\ \mathrm{Vertex}\ \mathrm{n}) \\
\qquad\qquad (\mathrm{Indices} : \mathrm{Buffer}\ (\mathrm{m} : \mathrm{Nat}\ \mathbf{where}\ \mathrm{m} < \mathrm{n})) \\
\qquad\qquad (\mathrm{m} : \mathrm{Matrix}) \\
\quad : \mathrm{Vector}\ \mathrm{Vertex}\ (\mathrm{Indices.length}) = \\
\quad [\mathrm{Transform}\ (\mathrm{m}, \mathrm{Vertices}[\mathrm{i}])\ \mathbf{where}\ \mathrm{i} \leftarrow \mathrm{Indices}]
\end{array}$$

Better types!

quantification over a natural number

types do not admit null values

vector has an explicit length

$$
\begin{aligned}
&\mathsf{Transform}\ \{\boxed{\mathsf{n}} : \mathsf{Nat}\}\\
&\qquad\qquad(\mathsf{Vertices} : \boxed{\mathsf{Vector}}\ \mathsf{Vertex}\ \boxed{\mathsf{n}})\\
&\qquad\qquad(\mathsf{Indices} : \mathsf{Buffer}\ (\mathsf{m} : \mathsf{Nat}\ \textbf{where}\ \mathsf{m} < \mathsf{n}))\\
&\qquad\qquad(\mathsf{m} : \mathsf{Matrix})\\
&\quad : \mathsf{Vector}\ \mathsf{Vertex}\ (\mathsf{Indices.length}) =\\
&\quad [\mathsf{Transform}\ (\mathsf{m}, \mathsf{Vertices}[\mathsf{i}])\ \textbf{where}\ \mathsf{i} \leftarrow \mathsf{Indices}]
\end{aligned}
$$

Better types!

quantification over a natural number

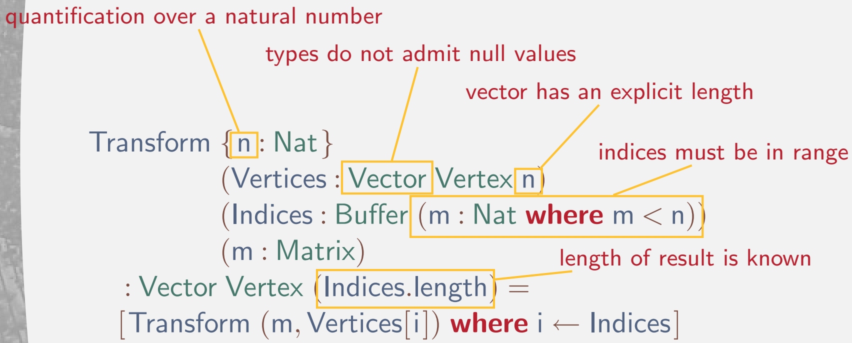
types do not admit null values

vector has an explicit length

indices must be in range

$$\begin{aligned}
&\mathsf{Transform}\ \{\boxed{\mathsf{n}} : \mathsf{Nat}\} \\
&\qquad\quad (\mathsf{Vertices} : \boxed{\mathsf{Vector}}\ \mathsf{Vertex}\ \boxed{\mathsf{n}}) \\
&\qquad\quad (\mathsf{Indices} : \mathsf{Buffer}\ \boxed{(\mathsf{m} : \mathsf{Nat}\ \textbf{where}\ \mathsf{m} < \mathsf{n})}) \\
&\qquad\quad (\mathsf{m} : \mathsf{Matrix}) \\
&\ : \mathsf{Vector}\ \mathsf{Vertex}\ (\mathsf{Indices.length}) = \\
&\ [\mathsf{Transform}\ (\mathsf{m}, \mathsf{Vertices}[\mathsf{i}])\ \textbf{where}\ \mathsf{i} \leftarrow \mathsf{Indices}]
\end{aligned}$$

Better types!



quantification over a natural number

types do not admit null values

vector has an explicit length

indices must be in range

length of result is known

```
Transform {n : Nat}
          (Vertices : Vector Vertex n)
          (Indices : Buffer (m : Nat where m < n))
          (m : Matrix)
  : Vector Vertex (Indices.length) =
  [Transform (m, Vertices[i]) where i ← Indices]
```

Better types!

quantification over a natural number

types do not admit null values

vector has an explicit length

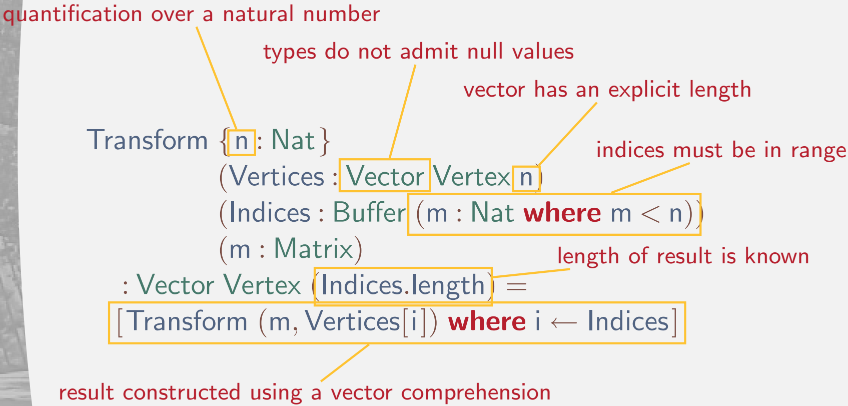
indices must be in range

```
Transform {n : Nat}
          (Vertices : Vector Vertex n)
          (Indices : Buffer (m : Nat where m < n))
          (m : Matrix)
  : Vector Vertex (Indices.length) =
  [Transform (m, Vertices[i]) where i ← Indices]
```

length of result is known

result constructed using a vector comprehension

Better types!



quantification over a natural number

types do not admit null values

vector has an explicit length

indices must be in range

```
Transform {n : Nat}
          (Vertices : Vector Vertex n)
          (Indices : Buffer (m : Nat where m < n))
          (m : Matrix)
  : Vector Vertex (Indices.length) =
  [Transform (m, Vertices[i]) where i ← Indices]
```

length of result is known

result constructed using a vector comprehension

Note that we mix terms (here: natural numbers) with types.

Dependent types

$A \rightarrow B$

Dependent types

$$A \to B$$

$$\begin{array}{l}(x\ : A)\ \to B(x)\\ \{x : A\} \to B(x)\end{array}$$

Dependent types

$$\mathsf{A} \rightarrow \mathsf{B}$$

$$\begin{array}{l} (\mathsf{x} \;:\mathsf{A}) \;\rightarrow \mathsf{B}(\mathsf{x}) \\ \{\mathsf{x} : \mathsf{A}\} \rightarrow \mathsf{B}(\mathsf{x}) \end{array}$$

$$(\mathsf{Indices} : \mathsf{Buffer\ Nat}) \rightarrow \mathsf{Vector\ Vertex}\ (\mathsf{Indices.length})$$

Type checking with dependent types

Type checker must perform evaluation.

Type checking with dependent types

Type checker must perform evaluation.

Is Vector Vertex $(2+2)$ the same as Vector Vertex 4?

Type checking with dependent types

Type checker must perform evaluation.

Is $\mathsf{Vector}\ \mathsf{Vertex}\ (2 + 2)$ the same as $\mathsf{Vector}\ \mathsf{Vertex}\ 4$?

Is $\mathsf{Vector}\ \mathsf{Vertex}\ (\mathsf{n} + 2)$ the same as $\mathsf{Vector}\ \mathsf{Vertex}\ (2 + \mathsf{n})$?

The power of dependent types

We can add near-arbitrary properties and restrictions to our types:

The power of dependent types

We can add near-arbitrary properties and restrictions to our types:

- Vectors of a certain length
- Numbers in a certain range

The power of dependent types

We can add near-arbitrary properties and restrictions to our types:

- Vectors of a certain length
- Numbers in a certain range
- Sorted lists; lists of even numbers; lists without duplicates

The power of dependent types

We can add near-arbitrary properties and restrictions to our types:

- Vectors of a certain length
- Numbers in a certain range
- Sorted lists; lists of even numbers; lists without duplicates
- Associative and commutative binary operators

The power of dependent types

We can add near-arbitrary properties and restrictions to our types:

- Vectors of a certain length
- Numbers in a certain range
- Sorted lists; lists of even numbers; lists without duplicates
- Associative and commutative binary operators
- Properly escaped OS commands
- Well-formed SQL queries
- . . .

Strong types are helpful

- We can make illegal configurations impossible to represent.
- We can preserve information we obtain from run-time testing.
- With a suitable development environment, types can guide the programming process.

Curry-Howard correspondence

Programming is like reasoning in (intuitionistic) logic

property	type
proof	program

Curry-Howard correspondence

Programming is like reasoning in (intuitionistic) logic

property proof	type program
truth	inhabited type
falsity	uninhabited type
conjunction	pair
disjunction	union type
implication	function
negation	function to the uninhabited type
universal quantification	dependent function
existential quantification	dependent pair

Universe constructions

A particular strength of dependently typed systems is that we can compute types from values.

Universe constructions

A particular strength of dependently typed systems is that we can compute types from values.

A type of codes together with an interpretation function mapping codes to types is called a universe.

Universe constructions

A particular strength of dependently typed systems is that we can compute types from values.

A type of codes together with an interpretation function mapping codes to types is called a universe.

Because we can analyze the codes as normal values, we can write functions that are extremely generic by using a universe.

Simple universe example: C-style printf

Codes

Format strings such as `"Test(%s): %d"`.

Interpretation function

Function that maps a format string to the type of the corresponding printf function, such as $(\text{String}, \text{Int}) \rightarrow \text{String}$.

Using the universe

We can define printf as an ordinary function.

Advanced universe example: databases

The “untyped” approach

Construct SQL queries as strings and send them to the database.

The (E)DSL approach

Have a special language mechanism or library to construct syntactically correct SQL queries.

The model-driven approach

Take the schema of a database and generate suitable datatypes and interface code from it. Then use the generated code.

The databases universe

Code

The database schemas.

Interpretation function

Takes schemas and computes suitable datatypes.

Using the universe

Together with an EDSL, we can write type-safe queries that are guaranteed to adhere to the schema, and can adapt when the schema changes.

Does it work yet?

Dependently typed programming languages are currently in an experimental stage:

- Good enough to write smaller programs.
- Lack of libraries.
- Not yet very optimized for performance.
- Quite a number of interesting and challenging problems that we can solve.
- If you want to try a language: check out Agda.
- Haskell allows a limited encoding of dependent types.