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.

```csharp
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.

```csharp
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.

```csharp
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.

```csharp
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.

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

```csharp
 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?
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?

```java
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;
}
```
Better types!

$$\text{Transform } \{ n : \text{Nat} \}$$

$$(\text{Vertices} : \text{Vector Vertex } n)$$

$$(\text{Indices} : \text{Buffer (} m : \text{Nat where } m < n))$$

$$(m : \text{Matrix})$$

$$: \text{Vector Vertex (Indices.length)} =$$

$$[\text{Transform (} m, \text{Vertices}[i]) \text{ where } i \leftarrow \text{Indices}]$$
Better types!

types do not admit null values

\[
\text{Transform } \{ n : \text{Nat} \} \\
\text{(Vertices : Vector Vertex n)} \\
\text{(Indices : Buffer (m : Nat \text{ where } m < n))} \\
\text{(m : Matrix)} \\
: \text{Vector Vertex (Indices.length)} = \\
[\text{Transform (m, Vertices[i]) \text{ where } i \leftarrow \text{Indices}}]
\]
Better types!

quantification over a natural number

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

quantification over a natural number

types do not admit null values

\[
\text{vector has an explicit length} \\
\text{indices must be in range} \\
\text{length of result is known} \\
\text{result constructed using a vector comprehension}
\]

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

quantification over a natural number

\[
\text{Transform } \{n\} : \text{Nat} \\
(\text{Vertices} : \text{Vector} \text{Vertex} n) \\
(\text{Indices} : \text{Buffer} (m : \text{Nat} \text{ where } m < n)) \\
(m : \text{Matrix})
\]

: Vector Vertex (Indices.length) =

\[
[\text{Transform} (m, \text{Vertices}[i]) \text{ where } i \leftarrow \text{Indices}]
\]

types do not admit null values

vector has an explicit length

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

Quantification over a natural number

types do not admit null values

vector has an explicit length

indices must be in range

Transform \( \{ n : \text{Nat} \} \)

(Vertices : \text{Vector} \text{Vertex} \( n \))

(Indices : \text{Buffer} (m : \text{Nat} \textbf{where} m < n))

(m : \text{Matrix})

: \text{Vector} \text{Vertex} (\text{Indices}.\text{length}) =

[Transform (m, Vertices[i]) \textbf{where} i \leftarrow \text{Indices}]
Better types!

quantification over a natural number

types do not admit null values

Transform \( \{ n : \text{Nat} \} \)

(Vertices : Vector Vertex \( n \))

(Indices : Buffer \( (m : \text{Nat} \text{ where } m < n) \))

(m : Matrix)

: Vector Vertex \( \text{Indices.length} \) =

[Transform \( (m, \text{Vertices}[i]) \text{ where } i \leftarrow \text{Indices} \)]

- vector has an explicit length
- indices must be in range
- length of result is known
- result constructed using a vector comprehension

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

quantification over a natural number

Transform \( \{ n : \text{Nat} \} \)
(Vertices : Vector \( n \))
(Indices : Buffer \( (m : \text{Nat} \text{ where } m < n) \))
(m : Matrix)

: Vector \( n \) \( \text{Vertex} \)

\[
\text{length} = \text{Transform} \ (m, \text{Vertices}[i]) \text{ where } i \leftarrow \text{Indices}
\]

result constructed using a vector comprehension

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

- types do not admit null values
- vector has an explicit length
- indices must be in range
- length of result is known

quantification over a natural number
Better types!

quantification over a natural number

types do not admit null values

Transform \( \{ n : \text{Nat} \} \)
(Vertices : Vector Vertex \( n \))
(Indices : Buffer \( (m : \text{Nat \ where} \ m < n) \))
(m : Matrix)

: Vector Vertex \( \text{Indices.length} \) =
[Transform \( (m, \text{Vertices}[i]) \) \textbf{where} \( i \leftarrow \text{Indices} \)]

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

\[ A \rightarrow B \]
Dependent types

\[ A \rightarrow B \]

\[(x : A) \rightarrow B(x)\]

\[\{x : A\} \rightarrow B(x)\]
Dependent types

\[ A \rightarrow B \]

\[(x : A) \rightarrow B(x) \]
\[ \{x : A\} \rightarrow B(x) \]

(Indices : Buffer Nat) \rightarrow \text{Vector Vertex (Indices.length)}
Type checking with dependent types

Type checker must perform evaluation.
Type checking with dependent types

Type checker must perform evaluation.

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

Is \( \text{Vector Vertex} (n + 2) \) the same as \( \text{Vector Vertex} (2 + n) \)?
Type checking with dependent types

Type checker must perform evaluation.

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

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

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proof</td>
<td>Program</td>
</tr>
<tr>
<td>Truth</td>
<td>Inhabited type</td>
</tr>
<tr>
<td>Falsity</td>
<td>Uninhabited type</td>
</tr>
<tr>
<td>Conjunction</td>
<td>Union type</td>
</tr>
<tr>
<td>Implication</td>
<td>Function</td>
</tr>
<tr>
<td>Negation</td>
<td>Function to the uninhabited type</td>
</tr>
<tr>
<td>Universal quantification</td>
<td>Dependent function</td>
</tr>
<tr>
<td>Existential quantification</td>
<td>Dependent pair</td>
</tr>
</tbody>
</table>
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 `(String, Int) → 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.