Type parameterization

Or: reason #213 why Java is terrible at everything

Koz Ross

4th May, 2017

[AUT CSC]

AUT Computer Science Club

Outline

What is this, and why should I care?

How do we implement this?

How good are these?

There's this one thing I don't get...

Example 1

Consider the following function:

Example 1

Consider the following function:

```
int add () {
  return 2 + 3;
}
```

Example 1

Consider the following function:

```
int add () {
  return 2 + 3;
}
```

This function is very inflexible — it might as well be a constant!

Example 1

We can make the function more flexible (and thus, do more work) by *parameterizing* its arguments.

Example 1

We can make the function more flexible (and thus, do more work) by *parameterizing* its arguments. That way, the *user* can decide what numbers it gets to add instead of us:

Example 1

We can make the function more flexible (and thus, do more work) by *parameterizing* its arguments. That way, the *user* can decide what numbers it gets to add instead of us:

```
int add (int x, int y) {
  return x + y;
}
```

Example 1

We can make the function more flexible (and thus, do more work) by *parameterizing* its arguments. That way, the *user* can decide what numbers it gets to add instead of us:

```
int add (int x, int y) {
  return x + y;
}
```

We call this *argument parameterization*, and it is a very useful thing to have (in fact, programming would be pretty pointless without it).

Example 1

We can make the function more flexible (and thus, do more work) by *parameterizing* its arguments. That way, the *user* can decide what numbers it gets to add instead of us:

```
int add (int x, int y) {
  return x + y;
}
```

We call this *argument parameterization*, and it is a very useful thing to have (in fact, programming would be pretty pointless without it). But then, what if our user wants to add two `floats` instead?

Example 2

Consider this (partial) definition of a (singly) linked list:

Example 2

Consider this (partial) definition of a (singly) linked list:

```
struct node {
  int data;
  node* next;
};

struct list {
  node* first;
};

list* list_new ();
```

Example 2

Consider this (partial) definition of a (singly) linked list:

```
struct node {
  int data;
  node* next;
};

struct list {
  node* first;
};

list* list_new ();
```

What if our user wanted a list of `floats` instead?

Example 2

Consider this (partial) definition of a (singly) linked list:

```
struct node {
  int data;
  node* next;
};

struct list {
  node* first;
};

list* list_new ();
```

What if our user wanted a list of `floats` instead? Does it really matter for list operations what kind of data we're storing?

The problem

These are both examples of *type restrictions*.

The problem

These are both examples of *type restrictions*. Normally, these are a *good* thing, as we wouldn't want something like this to compile:

The problem

These are both examples of *type restrictions*. Normally, these are a *good* thing, as we wouldn't want something like this to compile:

```
/* this won't compile */
sqrt("10");
/* but will *run* in JavaScript, sigh... */
```

The problem

These are both examples of *type restrictions*. Normally, these are a *good* thing, as we wouldn't want something like this to compile:

```
/* this won't compile */
sqrt("10");
/* but will *run* in JavaScript, sigh... */
```

However, in our two examples, type restrictions get in our way.

The problem

These are both examples of *type restrictions*. Normally, these are a *good* thing, as we wouldn't want something like this to compile:

```
/* this won't compile */
sqrt("10");
/* but will *run* in JavaScript, sigh... */
```

However, in our two examples, type restrictions get in our way. Wouldn't it be nice if we could parameterize over *types* as well?

What would this look like?

What would this look like?

```
struct node <T> {
    T data;
    node <T>* next;
  };

  struct list <T> {
    node <T>* first;
  };

  list <T>* list_new ();
```

What would this look like?

```
struct node <T> {
    T data;
    node <T>* next;
  };

  struct list <T> {
    node <T>* first;
  };

  list <T>* list_new ();
```

So now, if the user wants a list of `int`s, they will write
`list <int>* foo = list_new();`.
If they prefer a list of `float`s, they can write
`list <float>* bar = list_new();`.

Why do we care?

Why do we care?

- Write the code for a list *once*, and it will be able to store anything a user could want.

Why do we care?

- Write the code for a list *once*, and it will be able to store anything a user could want.
- Still have protection against things like this:

```
list <int>* foo = list_new();
/* won't compile */
list_insert(foo, "bar");
```

Why do we care?

- Write the code for a list *once*, and it will be able to store anything a user could want.
- Still have protection against things like this:

```
list <int>* foo = list_new();
/* won't compile */
list_insert(foo, "bar");
```

- Can write very useful things:

```
struct pair <T,U> {
  T left;
  U right;
}; /* a pair of anything! */
```

Why do we care?

- Write the code for a list *once*, and it will be able to store anything a user could want.
- Still have protection against things like this:

```
list <int>* foo = list_new();
/* won't compile */
list_insert(foo, "bar");
```

- Can write very useful things:

```
struct pair <T,U> {
  T left;
  U right;
}; /* a pair of anything! */
```

In short: Type parameterization makes our code more flexible, more concise, and generally better.

Some terminology

A *type parameter* is a placeholder type.

Some terminology

A *type parameter* is a placeholder type. When we mention it in a definition (like a structure or a function), we call this a *declaration*:

Some terminology

A *type parameter* is a placeholder type. When we mention it in a definition (like a structure or a function), we call this a *declaration*:

```
struct pair <T,U> { /* declaration */
  T left;
  U right;
}
```

Some terminology

A *type parameter* is a placeholder type. When we mention it in a definition (like a structure or a function), we call this a *declaration*:

```
struct pair <T,U> { /* declaration */
  T left;
  U right;
}
```

Later, when we actually use the structure of function, we have to provide an actual type for the type parameter (*instantiation*):

Some terminology

A *type parameter* is a placeholder type. When we mention it in a definition (like a structure or a function), we call this a *declaration*:

```
struct pair <T,U> { /* declaration */
  T left;
  U right;
}
```

Later, when we actually use the structure of function, we have to provide an actual type for the type parameter (*instantiation*):

```
pair <float, int> foo; /* instantiation */
```

Homogenous translation

Homogenous translation

- Type parameter *declarations* (and anything using them) get promoted to some specific type and compiled on the spot.

Homogenous translation

- Type parameter *declarations* (and anything using them) get promoted to some specific type and compiled on the spot.
- Type parameter *instantiations* are first checked for consistency; if no problems are found, their types are simply ignored where appropriate.

Homogenous translation

- Type parameter *declarations* (and anything using them) get promoted to some specific type and compiled on the spot.
- Type parameter *instantiations* are first checked for consistency; if no problems are found, their types are simply ignored where appropriate.

Thus, at *runtime*, a `pair <int, float>` is no different to a `pair <float, char*>` — might as well be `pair <wtf, wtf>` for all we care.

Heterogenous translation

Heterogenous translation

- Type parameter *declarations* (and anything using them) get turned into a ‘template’, with gaps where the type parameters should go. Nothing gets compiled yet.

Heterogenous translation

- Type parameter *declarations* (and anything using them) get turned into a ‘template’, with gaps where the type parameters should go. Nothing gets compiled yet.
- When the compiler sees a type parameter *instantiation*, it copy-pastes the types into the template, compiles the result, and uses the result for all identical future cases.

Heterogenous translation

- Type parameter *declarations* (and anything using them) get turned into a 'template', with gaps where the type parameters should go. Nothing gets compiled yet.
- When the compiler sees a type parameter *instantiation*, it copy-pastes the types into the template, compiles the result, and uses the result for all identical future cases.

This, when the compiler sees `pair<int, float>` for the first time, it will compile a special version for just those types; if it later sees `pair<float, char*>`, it'll compile a special version for those types; and so on.

Tradeoffs for homogenous translation

Tradeoffs for homogenous translation

Advantages

- Simple:

Tradeoffs for homogenous translation

Advantages

- Simple:
    - Compiles faster

Tradeoffs for homogenous translation

Advantages

- Simple:
  - Compiles faster
  - Simpler compiler logic

Tradeoffs for homogenous translation

Advantages

- Simple:
  - Compiles faster
  - Simpler compiler logic
- Smaller binaries:

Tradeoffs for homogenous translation

Advantages

- Simple:
  - Compiles faster
  - Simpler compiler logic
- Smaller binaries:
  - Less space required

Tradeoffs for homogenous translation

Advantages

- Simple:
  - Compiles faster
  - Simpler compiler logic
- Smaller binaries:
  - Less space required
  - Can use instruction cache effectively

Tradeoffs for homogenous translation

Advantages

- Simple:
  - Compiles faster
  - Simpler compiler logic
- Smaller binaries:
  - Less space required
  - Can use instruction cache effectively

Disadvantages

- No type information at runtime

Tradeoffs for homogenous translation

Advantages

- Simple:
  - Compiles faster
  - Simpler compiler logic
- Smaller binaries:
  - Less space required
  - Can use instruction cache effectively

Disadvantages

- No type information at runtime
- Indirection:

Tradeoffs for homogenous translation

Advantages

- Simple:
  - Compiles faster
  - Simpler compiler logic
- Smaller binaries:
  - Less space required
  - Can use instruction cache effectively

Disadvantages

- No type information at runtime
- Indirection:
  - Overhead for a pointer to the data

Tradeoffs for homogenous translation

Advantages

- Simple:
  - Compiles faster
  - Simpler compiler logic
- Smaller binaries:
  - Less space required
  - Can use instruction cache effectively

Disadvantages

- No type information at runtime
- Indirection:
  - Overhead for a pointer to the data
  - Extra pointer chasing

Tradeoffs for heterogenous translation

Tradeoffs for heterogenous translation

Advantages

- Type information available at runtime

Tradeoffs for heterogenous translation

Advantages

- Type information available at runtime
- No extra indirection

Tradeoffs for heterogenous translation

Advantages

- Type information available at runtime
- No extra indirection

Disadvantages

- Bigger binaries:

Tradeoffs for heterogenous translation

Advantages

- Type information available at runtime
- No extra indirection

Disadvantages

- Bigger binaries:
  - More space needed

Tradeoffs for heterogenous translation

Advantages

- Type information available at runtime
- No extra indirection

Disadvantages

- Bigger binaries:
  - More space needed
  - No hope for instruction cache

Tradeoffs for heterogenous translation

Advantages

- Type information available at runtime
- No extra indirection

Disadvantages

- Bigger binaries:
  - More space needed
  - No hope for instruction cache
- Complex:

Tradeoffs for heterogenous translation

Advantages

- Type information available at runtime
- No extra indirection

Disadvantages

- Bigger binaries:
  - More space needed
  - No hope for instruction cache
- Complex:
  - Compiles slower

Tradeoffs for heterogenous translation

Advantages

- Type information available at runtime
- No extra indirection

Disadvantages

- Bigger binaries:
  - More space needed
  - No hope for instruction cache
- Complex:
  - Compiles slower
  - More complex compiler logic

So which is better?

So which is better?

- No single answer — both have various tradeoffs in general

So which is better?

- No single answer — both have various tradeoffs in general
- Need to be viewed in the context of a particular language

So which is better?

- No single answer — both have various tradeoffs in general
- Need to be viewed in the context of a particular language
- Let's see some examples!

So which is better?

- No single answer — both have various tradeoffs in general
- Need to be viewed in the context of a particular language
- Let's see some examples!

We will have rating indicators:

So which is better?

- No single answer — both have various tradeoffs in general
- Need to be viewed in the context of a particular language
- Let's see some examples!

We will have rating indicators:

✓

'This is good (or not a problem)!'

So which is better?

- No single answer — both have various tradeoffs in general
- Need to be viewed in the context of a particular language
- Let's see some examples!

We will have rating indicators:

✓

'This is good (or not a problem)!'

✗

'This is bad (or a *real* problem)!'

Homogenous translation done well: C

Homogenous translation done well: C

Honesty note: C doesn't technically have homogenous translation built-in. We have to fake it with `void*`.

Homogenous translation done well: C

Honesty note: C doesn't technically have homogenous translation built-in. We have to fake it with `void*`.

Advantages

- Simple

Homogenous translation done well: C

Honesty note: C doesn't technically have homogenous translation built-in. We have to fake it with `void*`.

Advantages

- Simple ✓

Homogenous translation done well: C

Honesty note: C doesn't technically have homogenous translation built-in. We have to fake it with `void*`.

Advantages

- Simple ✓
- Smaller binaries

Homogenous translation done well: C

Honesty note: C doesn't technically have homogenous translation built-in. We have to fake it with `void*`.

Advantages

- Simple ✓
- Smaller binaries ✓

Homogenous translation done well: C

Honesty note: C doesn't technically have homogenous translation built-in. We have to fake it with `void*`.

Advantages

- Simple ✓
- Smaller binaries ✓

Disadvantages

- No type information at runtime

Homogenous translation done well: C

Honesty note: C doesn't technically have homogenous translation built-in. We have to fake it with `void*`.

Advantages

- Simple ✓
- Smaller binaries ✓

Disadvantages

- No type information at runtime ✓

Homogenous translation done well: C

Honesty note: C doesn't technically have homogenous translation built-in. We have to fake it with `void*`.

Advantages

- Simple ✓
- Smaller binaries ✓

Disadvantages

- No type information at runtime ✓
- Indirection

Homogenous translation done well: C

Honesty note: C doesn't technically have homogenous translation built-in. We have to fake it with `void*`.

Advantages

- Simple ✓
- Smaller binaries ✓

Disadvantages

- No type information at runtime ✓
- Indirection ✗

Homogenous translation done well: C

Honesty note: C doesn't technically have homogenous translation built-in. We have to fake it with `void*`.

Advantages

- Simple ✓
- Smaller binaries ✓

Disadvantages

- No type information at runtime ✓
- Indirection ✗

Overall verdict: $\frac{3}{4}$

Homogenous translation done well: C

Honesty note: C doesn't technically have homogenous translation built-in. We have to fake it with `void*`.

Advantages

- Simple ✓
- Smaller binaries ✓

Disadvantages

- No type information at runtime ✓
- Indirection ✗

Overall verdict: $\frac{3}{4}$ (a B).

Homogenous translation done badly: Java

Homogenous translation done badly: Java

Advantages

- Simple

Homogenous translation done badly: Java

Advantages

- Simple ✗

Homogenous translation done badly: Java

Advantages

- Simple ✗
- Smaller binaries

Homogenous translation done badly: Java

Advantages

- Simple ✗
- Smaller binaries ✗

Homogenous translation done badly: Java

Advantages

- Simple ✗
- Smaller binaries ✗

Disadvantages

- No type information at runtime

Homogenous translation done badly: Java

Advantages

- Simple ✗
- Smaller binaries ✗

Disadvantages

- No type information at runtime ✗

Homogenous translation done badly: Java

Advantages

- Simple ✗
- Smaller binaries ✗

Disadvantages

- No type information at runtime ✗
- Indirection

Homogenous translation done badly: Java

Advantages

- Simple ✗
- Smaller binaries ✗

Disadvantages

- No type information at runtime ✗
- Indirection ✓

Homogenous translation done badly: Java

Advantages

- Simple ✗
- Smaller binaries ✗

Overall verdict: $\frac{1}{4}$

Disadvantages

- No type information at runtime ✗
- Indirection ✓

Homogenous translation done badly: Java

Advantages

- Simple ✗
- Smaller binaries ✗

Disadvantages

- No type information at runtime ✗
- Indirection ✓

Overall verdict: $\frac{1}{4}$ (drop out of university Java, you're drunk)

Heterogenous translation done well: C#

Heterogenous translation done well: C#

Advantages

- Type information available at runtime

Heterogenous translation done well: C#

Advantages

- Type information available at runtime ✓

Heterogenous translation done well: C#

Advantages

- Type information available at runtime ✓
- No extra indirection

Heterogenous translation done well: C#

Advantages

- Type information available at runtime ✓
- No extra indirection ✓

Heterogenous translation done well: C#

Advantages

- Type information available at runtime ✓
- No extra indirection ✓

Disadvantages

- Bigger binaries

Heterogenous translation done well: C#

Advantages

- Type information available at runtime ✓
- No extra indirection ✓

Disadvantages

- Bigger binaries ✓

Heterogenous translation done well: C#

Advantages

- Type information available at runtime ✓
- No extra indirection ✓

Disadvantages

- Bigger binaries ✓
- Complex

Heterogenous translation done well: C#

Advantages

- Type information available at runtime ✓
- No extra indirection ✓

Disadvantages

- Bigger binaries ✓
- Complex ✓

Heterogenous translation done well: C#

Advantages

- Type information available at runtime ✓
- No extra indirection ✓

Disadvantages

- Bigger binaries ✓
- Complex ✓

Overall verdict: $\frac{4}{4}$

Heterogenous translation done well: C#

Advantages

- Type information available at runtime ✓
- No extra indirection ✓

Disadvantages

- Bigger binaries ✓
- Complex ✓

Overall verdict: $\frac{4}{4}$ (I swear that Microsoft didn't pay me!)

Heterogenous translation done badly: C++

Heterogenous translation done badly: C++

Advantages

- Type information available at runtime

Heterogenous translation done badly: C++

Advantages

- Type information available at runtime ✗

Heterogenous translation done badly: C++

Advantages

- Type information available at runtime ✗
- No extra indirection

Heterogenous translation done badly: C++

Advantages

- Type information available at runtime ✗
- No extra indirection ✓

Heterogenous translation done badly: C++

Advantages

- Type information available at runtime ✗
- No extra indirection ✓

Disadvantages

- Bigger binaries

Heterogenous translation done badly: C++

Advantages

- Type information available at runtime ✗
- No extra indirection ✓

Disadvantages

- Bigger binaries ✗

Heterogenous translation done badly: C++

Advantages

- Type information available at runtime ✗
- No extra indirection ✓

Disadvantages

- Bigger binaries ✗
- Complex

Heterogenous translation done badly: C++

Advantages

- Type information available at runtime ✗
- No extra indirection ✓

Disadvantages

- Bigger binaries ✗
- Complex ✗

Heterogenous translation done badly: C++

Advantages

- Type information available at runtime ✗
- No extra indirection ✓

Disadvantages

- Bigger binaries ✗
- Complex ✗

Overall verdict: $\frac{1}{4}$

Heterogenous translation done badly: C++

Advantages

- Type information available at runtime ✗
- No extra indirection ✓

Disadvantages

- Bigger binaries ✗
- Complex ✗

Overall verdict: $\frac{1}{4}$ (bad idea in the 80s, bad idea now)

Conclusion

Conclusion

- Type parametrization is something we want (and language designers have obliged)

Conclusion

- Type parametrization is something we want (and language designers have obliged)
- There's more than one way to do it, and it must be viewed in the context of the language they inhabit

Conclusion

- Type parametrization is something we want (and language designers have obliged)
- There's more than one way to do it, and it must be viewed in the context of the language they inhabit
- More work is still being done on this!

Conclusion

- Type parametrization is something we want (and language designers have obliged)
- There's more than one way to do it, and it must be viewed in the context of the language they inhabit
- More work is still being done on this!
- Important to understand how something works (don't just blindly follow hype and buzzwords)

Conclusion

- Type parametrization is something we want (and language designers have obliged)
- There's more than one way to do it, and it must be viewed in the context of the language they inhabit
- More work is still being done on this!
- Important to understand how something works (don't just blindly follow hype and buzzwords)

"In software development, abstraction is often used as a synonym for indirection. Not so in mathematics."

Susan Potter (`@SusanPotter`)

Question time!

?