Sixten: Unboxed functional programming

Olle Fredriksson

Mødegruppe for F#unktionelle Københavnere

2020-08-25

Sixten is a functional language with dependent types and **unboxed data by default**.

Today:

- Compiling polymorphism
    - Monomorphisation
    - Uniform representation
    - Intensional polymorphism
    - **Representation polymorphism**
- Some live coding
- Where Sixten is at right now
- Where Sixten is going

```
map f Nil = Nil
map f (Cons x xs) = Cons (f x) (map f xs)
```

```
map : (Int -> Int) -> List Int -> List Int
map f Nil = Nil
map f (Cons x xs) = Cons (f x) (map f xs)
```

Polymorphism

```
map : forall a b. (a -> b) -> List a -> List b
map f Nil = Nil
map f (Cons x xs) = Cons (f x) (map f xs)
```

```
map : forall a b. (a -> b) -> List a -> List b

later =
  map (\x. x + 10) [1, 2, 3]

evenLater =
  map (\(MkTuple _ b). b) [MkTuple 1 False, MkTuple 2 True]
```

```haskell
data Tuple a b = MkTuple a b
data List a = Nil | Cons a (List a)
```

```
map : forall a b. (a -> b) -> List a -> List b

later =
  map @{a = Int, b = Int} (\x. x + 10) [1, 2, 3]

evenLater =
  map
    @{a = Tuple Int Bool, b = Bool}
    (\(MkTuple _ b). b)
    [MkTuple 1 False, MkTuple 2 True]
```

How is polymorphism compiled?

How is polymorphism compiled?

Uniform representation

or

Monomorphisation

Uniform representation

```
map : forall a b. (a -> b) -> List a -> List b
map f Nil = Nil
map f (Cons x xs) = Cons (f x) (map f xs)
```

```
word* map(word* f, word* list) {
  …
}
```

value	memory representation
`Nil`	`→[NIL_TAG]`
`Cons x xs`	`→[CONS_TAG, x, xs]`
`f`	`→[...]`

```c
word* map(word* f, word* list) {
  switch (list[0]) {
    case NIL_TAG:
      word* result = heap_alloc(sizeof(word));
      result[0] = NIL_TAG;
      return result;

    case CONS_TAG:
      word* x  = (word*) list[1];
      word* xs = (word*) list[2];
      word* result = heap_alloc(sizeof(word) * 3);
      result[0] = CONS_TAG;
      result[1] = (word) apply(f, x);
      result[2] = (word) map(f, xs);
      return result;
  }
}
```

Monomorphisation

```
map : forall a b. (a -> b) -> List a -> List b

later =
  map @{a = Int, b = Int} (\x. x + 10) [1, 2, 3]
```

```
map : forall a b. (a -> b) -> List a -> List b

map_Int_Int : (Int -> Int) -> List Int -> List Int

later =
  map_Int_Int (\x. x + 10) [1, 2, 3]
```

```
map : forall a b. (a -> b) -> List a -> List b

map_Int_Int : (Int -> Int) -> List Int -> List Int

later =
  map_Int_Int (\x. x + 10) [1, 2, 3]

evenLater =
  map
    @{a = Tuple Int Bool, b = Bool}
    (\(MkTuple _ b). b)
    [MkTuple 1 False, MkTuple 2 True]
```

```
map : forall a b. (a -> b) -> List a -> List b

map_Int_Int : (Int -> Int)  -> List Int -> List Int

later = map_Int_Int (\x. x + 10) [1, 2, 3]

map_Tuple_Int_Bool_Bool :
  (Tuple Int Bool -> Bool) -> List (Tuple Int Bool) -> List Bool

evenLater =
  map_Tuple_Int_Bool_Bool
    (\(MkTuple _ b). b)
    [MkTuple 1 False, MkTuple 2 True]
```

Pros and cons

Uniform representation	Monomorphisation
+ Compile once (modular)	- Compile for each type instantiation (anti-modular)
- Inefficient	+ Efficient (specialised implementations)
- Boxed data	+ Unboxed data
+ Advanced features (polymorphic recursion, existential types)	- No polymorphic recursion, no existential types

Sixten
+ Compile once (modular)
- Inefficient (but fixable?)
+ Unboxed data
+ Advanced features (polymorphic recursion, existential types)

Intensional polymorphism (Harper, Morrisett, 1994)

```
map : forall a b. (a -> b) -> List a -> List b
map f Nil = Nil
map f (Cons x xs) = Cons (f x) (map f xs)
```

```
word* map(type a, type b, word* f, word* list) {
  switch (list[0]) {
    case NIL_TAG:
      word* result = heap_alloc(sizeof(word));
      result[0] = NIL_TAG;
      return result;

    case CONS_TAG:
      word* x  = &list[1];
      word* xs = (word*) list[1 + type_size_in_words(a)];
      word* result = heap_alloc(sizeof(word) + type_size(b) + sizeof(word));
      result[0] = CONS_TAG;
      apply(f, a, x, &result[1]);
      result[1 + type_size_in_words(b)] = (word) map(a, b, f, xs);
      return result;
  }
}
```

Intrusive list:

```
[CONS_TAG, x, *]--->[CONS_TAG, y, *]--->[NIL_TAG]
```

Non-intrusive lists:

```
           [x]                  [y]
            ^                    ^
            |                    |
[CONS_TAG, *, *]--->[CONS_TAG, *, *]--->[NIL_TAG]
```

Sixten:

```
boxed
data List a = Nil | Cons a (List a)
```

What are types at runtime?

```
data Type = Function Type Type | Product Type Type | ...

evenLater =
  map
    @{a = Product Int Bool, b = Bool}
    (\(MkTuple _ b). b)
    [MkTuple 1 False, MkTuple 2 True]
```

Intensional polymorphism

```
word* map(type a, type b, word* f, word* list) {
  switch (list[0]) {
    case NIL_TAG:
      word* result = heap_alloc(sizeof(word));
      result[0] = NIL_TAG;
      return result;

    case CONS_TAG:
      word* x  = &list[1];
      word* xs = (word*) list[1 + type_size_in_words(a)];
      word* result = heap_alloc(sizeof(word) + type_size(b) + sizeof(word));
      result[0] = CONS_TAG;
      apply(f, type_size(a), x, &result[1]);
      result[1 + type_size_in_words(b)] = (word) map(a, b, f, xs);
      return result;
  }
}
```

Representation polymorphism

```
word* map(size_t a_size, size_t b_size, word* f, word* list) {
  switch (list[0]) {
    case NIL_TAG:
      word* result = heap_alloc(sizeof(word));
      result[0] = NIL_TAG;
      return result;

    case CONS_TAG:
      word* x  = &list[1];
      word* xs = (word*) list[1 + a_size / sizeof(word)];
      word* result = heap_alloc(sizeof(word) + b_size + sizeof(word));
      result[0] = CONS_TAG;
      apply(f, a_size, x, &result[1]);
      result[1 + b_size / sizeof(word)] = (word) map(a_size, b_size, f, xs);
      return result;
  }
}
```

```
evenLater =
  map
    @{a = sizeof(Int) + sizeof(Bool), b = sizeof(Bool)}
    (\(MkTuple _ b). b)
    [MkTuple 1 False, MkTuple 2 True]
```

```
boxed
data BoxedMaybe a = BoxedNothing | BoxedJust a
data Maybe a = Nothing | Just a
```

```
sizeof(BoxedMaybe a) = sizeof(word*) = 8 bytes
sizeof(Maybe a) = max(sizeof(word), sizeof(word) + sizeof(a)) = sizeof(word) + sizeof(a)
```

value	memory representation
`BoxedNothing`	`→[BOXED_NOTHING_TAG]`
`BoxedJust x`	`→[BOXED_JUST_TAG, x]`
`Nothing`	`[NOTHING_TAG, padding of size a]`
`Just x`	`[JUST_TAG, x]`

Existentials must be boxed

```
data Sigma (A : Type) (B : A -> Type) where
  MkSigma : (a : A) -> B a -> Sigma A B
```

Closures must be boxed

```
a -> b
```

Recursive data definitions need to be boxed

```
data List a = Nil | Cons a (List a)
```

```
sizeof(List a) = sizeof(word*) + sizeof(a) + sizeof(List a)
```

Live coding unboxed immutable arrays

Erasure and quantitative types

```
map (A : Type) (B : Type) (f : A -> B) (xs : List A) =
  case xs of
    Nil -> Nil
    Cons x xs' -> Cons (f x) (map A B f xs')
```

Status

Original Sixten

- Slow type checker and compiler
- Buggy
- Type classes, extern code, implicit parameters, pi types, inductive families
- Compiles to LLVM
- Uses the conservative Boehm garbage collector
- Query-based
- Errors and hover-only language server

https://github.com/ollef/sixten

Reimplementation: Sixty

- Fast type checker (normalisation by evaluation + "gluing" from András Kovács smalltt)
- Not quite as buggy
- Implicit parameters, pi types, inductive families
- Query-based
- Language server with errors, go to definition, type-based completion, find references, renaming, code lenses
- No compiler yet

https://github.com/ollef/sixty

Future

- Sixty to replace Sixten
- LLVM backend for Sixty
- Precise GC
- Optional monomorphisation
- Make it useful

Summary

- Compiling polymorphism
  - Monomorphisation
  - Uniform representation
  - Intensional polymorphism
  - **Representation polymorphism**
- How Sixten represents data

https://github.com/ollef/sixty

https://github.com/ollef/sixten

https://ollef.github.io/blog

@ollfredo

```
boxed
data Box a = MkBox a

data List a = Nil | Cons a (Box (List a))
```