What is Needle?

Needle is an object-oriented functional programming language with a multimethod-based OO system, and a static type system with parameterized types and substantial ML-style type inference.

The Static Typing Heresy

In exploratory programming:

- Bottom-up programming; design is discovered, not imposed
- Domain knowledge comes from trying to solve the problems
- Subproblems knitted together to build abstractions

The Static Typing Heresy

In exploratory programming:

- Bottom-up programming; design is discovered, not imposed
- Domain knowledge comes from trying to solve the problems
- Subproblems knitted together to build abstractions

Exploratory programming is easier in a statically typed language with generic functions!

What are generic functions and multimethods?

First, let's look at classes in Needle.

```
class Thing {} // define a root class

class Rock(Thing) {}
class Paper(Thing) {}
class Scissors(Thing) {}
```

What are generic functions and multimethods?

First, let's look at classes in Needle.

```
class Thing {} // define a root class

class Rock(Thing) {}
class Paper(Thing) {}
class Scissors(Thing) {}
```

The curlies are simply where you define members:

```
class Cons[a] (List) {
  head a;
  tail List[a];
}
```

What are generic functions and multimethods?

```
generic beats? (Thing, Thing) -> Boolean;

method beats? (x Rock, y Scissors) { true; }
method beats? (x Paper, y Rock) { true; }
method beats? (x Scissors, y Paper) { true; }
method beats? (x Thing, y Thing) { false; }
```

`beats?(rock, rock)` $\Rightarrow$ *false*

Generic Functions and OO programming

In traditional OO, adding new methods to a class is unmodular even if it's possible.

```
generic inflammable? Thing -> Boolean;

method inflammable? (x Thing) { false }
method inflammable? (x Paper) { true }
```

Generic Functions and Functional Programming

Higher-order functions easily parameterize over behavior, but they don't parameterize over similar data types very well.

In Scheme:

```
(map function sequence)        ;; for lists
(vector-map function sequence) ;; for vectors
(string-map function sequence) ;; for strings
```

In Needle:

```
generic map c < Sequence . (a -> b, c[a]) -> c[b];
```

Needle's Type System

- Inspired by Bourdoncle and Merz's ML-sub (1997) and Bonniot (2001)

- Supports parametric types

- Supports type inference

Constrained Polymorphic Types

```
generic map c < Sequence . (a -> b, c[a]) -> c[b];
```

Type composed of two parts:

- ML-style type scheme
- Type constraints

Type scheme

A type scheme is:

- A nonpolymorphic class – Rock, Boolean
- any of a set of type variables – a, b, c
- A filled-in polymorphic class – List[List[Int]], a $\longrightarrow$ Boolean

Type constraints

Type constraints are conjunctions of subtype relationships; limit which types are permitted to satisfy the type variables in the type scheme.

```
generic negate a < Number . a -> a;
generic map c < Sequence . (a -> b, c[a]) -> c[b];

fun(seq) { map(negate, seq) }
// has type c < Sequence & a < Number . c[a] -> c[a]
```

How Type Inference Works

1. Top-down walk of each top-level expression's AST

2. Generate types of subexpressions, combining their constraint sets.

3. Verify the constraints are satisfiable

4. Simplify the constraints

Type Inference in Action

We do type inference on all code that isn't a generic declaration, or the argument specializer list on a method.

```
let foo = fun (x, f, g, seq) {
            if (x == 3) {
              map(f, seq);
            } else {
              map(fun (x) { g(g(x)) }, seq)
            }
          };
```

has inferred type:

```
  c < Sequence . (Integer, a -> a, a -> a, c[a]) -> c[a]
```

An Un-Simplified Type

The raw, unsimplified type of `foo`:

```
h < Number & (h, h) -> Boolean < (a, Integer) -> g & j < Sequence &
(k -> l, j[k]) -> j[l] < (b, d) -> i & n < Sequence &
(o -> p, n[o]) -> n[p] < (c, d) -> m &
(Boolean, f, f) -> f < (g, i, m) -> e . (a, b, c, d) -> e
```

After simplification:

```
c < Sequence . (Integer, a -> a, a -> a, c[a]) -> c[a]
```

Technical Commentary

- Generics support separate compilation
- Coverage/completeness tests independent of typechecking
- Everything but generics and method specializers have types inferred

Generic functions and Exploratory Programming, Take 2

How does the combination of type inference and generics really help?

Subproblems knitted together to build abstractions

Best way of composing subproblems is higher-order functions. Static typing helps here, because:

- Easier catch errors when the compiler fails fast
- Easier to discover common patterns when you can see the types
- Generic functions reduce necessary number of parameters

Domain knowledge comes from trying to solve problems

A type is a partial, approximate specification of a function.

Type inference means the compiler generate summaries for you.

Bottom-up style; design is discovered, not invented

- ML makes it hard to extend datatypes, but easy to write new behaviors

- Java makes it hard to add behaviors, but easy to extend datatypes

- Functions grow behavior; classes grow data

- For exploratory programming you need both

Future Work

- Improve type simplification
- Add dynamic loading
- Add interfaces

Interfaces

In current Needle, generic printing might have the interface:

```
generic print a -> String;

method print (s String) { s }
method print (b Boolean) { if (b) { "true" } else { "false" } }

method print (o a) { raise Error(); }
```

Throwing an exception hurts safety.

Interfaces, continued

What we want is something like this:

```
interface Print(a) {
  print a -> String;
}

generic print Print(a) . a -> String

String implements Print; // interfaces are added *post-hoc*
Boolean implements Print;

method print (s String) { s }
method print (b String) { if (b) { "true" } else { "false" } }
```

Interfaces

- Lets you add existing types to new protocols
- Fixes weakness of generic-function style – grouping methods.
- Idea stems from Haskell typeclasses.
- Implementation in progress.

How to get Needle

- Website at: `http://www.nongnu.org/needle`
- Mailing list at:

  `http://mail.nongnu.org/mailman/listinfo/needle-hackers`
- Email me at: `neelk@alum.mit.edu`