The Needle Programming Language

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.

What Is Needle For?

Needle is designed to support exploratory programming.

What Is Needle For?

Needle is designed to support exploratory programming.

1. Create extensible datatypes via subclassing

What Is Needle For?

Needle is designed to support exploratory programming.

1. Create extensible datatypes via subclassing

2. Use higher-order functions to build complex abstractions

What Is Needle For?

Needle is designed to support exploratory programming.

1. Create extensible datatypes via subclassing

2. Use higher-order functions to build complex abstractions

3. Use static typing to make “sloppy” coding easier

Literals and Identifiers

Usual literals:

```
3  'a'  "string"
```

C-like identifiers, augmented with `?` and `'`:

```
frob_foo  alphanumeric? x'
```

Function Calls

Supports most C/Java syntactic conventions as sugar for function calls:

```
f(3);                          // function call
range(from: 0, below: 10);     // with named args
x[3];                          // arrays: elt(x, 3)
x[3] = 5;                      // arrays: set_elt(x, 3, 5)
square.area;                   // sugar for: area(square)
square.area = 3;               // sugar for: set_area(square, 3)
```

Composite Expressions

Blocks:

```
{ foo(); bar(); 3 + baz() }
```

If-then-else:

```
if (foo?) { bar } else { frob(); baz }
```

Function expressions:

```
fun(x) { fun(y) { x + y } }
fun(vec, pos: x, offset: y) { vec[x] - y }
```

Bindings

Binding:

```
{ let y = 6;
  let f = fun(z) { z + y };  // y is captured
  f(2) // evals to 8
}
```

Local recursive functions:

```
{ rec fact(n) {
    if (n == 0) { 1 } else { n * fact(n - 1) }
  };
  fact(5)
}
```

Class Definitions

Classes have single inheritance and are multiply-rooted, with no root Object class.

```
class Point {
  constructor point;

  x Integer;
  y mutable Integer;
}

class ColorPoint (Point) {
  constructor colorpoint;

  color Color;
}
```

Constructors, Getters, and Setters

Making a Point object:

```
point(x: 3, y: 4)
colorpoint(x: 3, y: 4, color: red)
```

Accessing a Point:

```
{ let pt = point(xpos:3, ypos:4);
  pt.y = 9;
  pt.x + y(pt);
}
```

Polymorphic Classes

Classes also support *parametric polymorphism*:

```
class List[a] {}

class Nil[a] (List) {
  constructor nil;
}

class Cons[a] (List) {
  constructor cons;

  head a;
  tail List[a];
}
```

Generic Functions and Methods: The Example Hierarchy

First, let's set up a simple hierarchy for the examples:

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

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

Generic Functions and Methods: Multiple Dispatch

Generic functions enable method selection and multiple dispatch:

```
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];
```

Type Expressions

- Simple classes: `Integer`, `Boolean`, `Char`
- Type variables: `a`, `b`, `c`
- Parameterized classes: `List[a]`, `Table[Integer, Boolean]`
- Function types:
  - `Integer -> Boolean`
  - `(Integer, Integer) -> Integer`
  - `(String, start:Integer, len:Integer) -> String`

Polymorphic Constrained Types

Every expression's type consists of a type expression, plus a set of subtype constraints that the type variables have to satisfy:

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

fun(seq) { map(negate, seq) }
```

has type `c < Sequence & a < Number .  c[a] -> c[a]`

ML-sub

Needle's type system is:

- Based on Bourdoncle and Merz's ML-sub (1997)
- Supports type inference (Bonniot 2001)

Type Inference

Needle has type inference. Eg:

```
{ rec error_fact(n) {
     if (n == 0) { "1" } else { n * error_fact(n - 1) }
    };
    ...
}
```

The compiler will signal an error on this function.

The Type Inference Algorithm

The basic type inference algorithm has four steps:

1. Generate a polymorphic constrained type at each leaf in the AST.
2. Merge the types together, combining their constraint sets.
3. Check to see if the constraints have a solution. If there is no solution, then the expression has a type error.
4. Simplify the constraint set to report back to the user.

A Type Inference Example: Generating Leaf Types

```
generic (+) a < Number . (a, a) -> a;

fun(x) { x + x }
```

Let's see how types are assigned to each leaf.

A Type Inference Example: Generating Leaf Types

```
generic (+) a < Number . (a, a) -> a;

fun(x) { x + x }
```

Let's see how types are assigned to each leaf.

1. {} `fun(x) { x + x }`

A Type Inference Example: Generating Leaf Types

```
generic (+) a < Number . (a, a) -> a;

fun(x) { x + x }
```

Let's see how types are assigned to each leaf.

1.	{}	`fun(x) { x + x }`
2.	{x : t}	Abstract(t, `x + x`)

A Type Inference Example: Generating Leaf Types

```
generic (+) a < Number . (a, a) -> a;

fun(x) { x + x }
```

Let's see how types are assigned to each leaf.

1.	{}	`fun(x) { x + x }`
2.	{x : t}	Abstract(t, `x + x`)
3.	{x : t}	Abstract(t, Apply(`+`, `x`, `x`))

A Type Inference Example: Generating Leaf Types

```
generic (+) a < Number . (a, a) -> a;

fun(x) { x + x }
```

Let's see how types are assigned to each leaf.

1. {} `fun(x) { x + x }`
2. {x : t} Abstract(t, `x + x`)
3. {x : t} Abstract(t, Apply(+, x, x))
4. {x : t} Abstract(t, Apply(a < Number . (a,a) -> a, t, t))

A Type Inference Example: Merging Leaves

Oncae we have the type tree, we can merge the leaf types into a single constrained type:

1. Abstract(`t`, Apply(`a < Number .  (a,a) -> a`, `t`, `t`))

A Type Inference Example: Merging Leaves

Oncae we have the type tree, we can merge the leaf types into a single constrained type:

1. Abstract(`t`, Apply(`a < Number . (a,a) -> a`, `t`, `t`))

2. Abstract(`t`, `a < Number & (a,a) -> a < (t,t) -> b . b`)

A Type Inference Example: Merging Leaves

Oncae we have the type tree, we can merge the leaf types into a single constrained type:

1. Abstract(`t`, Apply(`a < Number . (a,a) -> a, t, t`))

2. Abstract(`t`, `a < Number & (a,a) -> a < (t,t) -> b . b`)

3. `a < Number & (a,a) -> a < (t,t) -> b . t -> b`

A Type Inference Example: Constraint Resolution

We must verify that there is at least one assignment to the variables that satisfies the constraints:

```
a < Number & (a,a) -> a < (t,t) -> b .  t -> b
```

Example: $\{$`t` $\leftarrow$ `a; b` $\leftarrow$ `a`$\}$

We check satisfiability using standard techniques:

- Compute the closure of the constraints.
- Run a satisfiability algorithm.

A Type Inference Example: Constraint Simplification

```
a < Number & (a,a) -> a < (t,t) -> b .  t -> b
```

is equivalent to

```
a < Number .  a -> a
```

For readability, inferred types must be simplified.

Comparison with ML

Pros:

- Datatypes can be extended with subclassing
- Generic functions give you controlled overloading

Cons:

- No principal types
- More complex type inference algorithm

Comparison with CLOS/Dylan

Pros:

- Integrates well with parametric polymorphism
- More precise types available for documentation

Cons:

- Stricter lambda-list rules

Future Work: 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.

Future Work: Interfaces, cont.

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 Boolean) { if (b) { "true" } else { "false" } }
```

Future Work: 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`