$\{\mathbf{FJ}_{\circ\mu}\}$

Safe, Flexible Recursive Types for Featherweight Java

Reuben N. S. Rowe

Department of Computing

Imperial College London

History

Java

Featherweight Java (FJ)

Nominal Types

`C`

`Object`

`Vector`

a weak and coarse-grained static type system

History

Java

Featherweight Java (FJ)

Nominal Types		Structural Types		Intersection Types
`C`				
`Object`	+	$\langle f{:}\sigma \rangle$ $\langle m{:}(\vec{\sigma}) \to \tau \rangle$	+	ω $\sigma \cap \tau$
`Vector`				

a very powerful and expressive static type system [3]

Roadblock

- The type system is not very *practical*:

```
class C extends Object
{
      C m() {return this;}
}
```

What types does the term `new C()` have?

Roadblock

- The type system is not very *practical*:

```
class C extends Object
{
      C m() {return this;}
}
```

  What types does the term `new C()` have?

  – `C`

Roadblock

- The type system is not very *practical*:

```
class C extends Object
{
      C m() {return this;}
}
```

What types does the term `new C()` have?

– `C`

– $\langle \mathtt{m} : () \to \mathtt{C} \rangle$

Roadblock

- The type system is not very *practical*:

```
class C extends Object
{
        C m() {return this;}
}
```

What types does the term `new C()` have?

– `C`

– $\langle \mathtt{m} : () \rightarrow \mathtt{C} \rangle$

– $\langle \mathtt{m} : () \rightarrow \langle \mathtt{m} : () \rightarrow \mathtt{C} \rangle \rangle$

Roadblock

- The type system is not very *practical*:

```
class C extends Object
{
        C m() {return this;}
}
```

What types does the term `new C()` have?

- $\mathtt{C}$
- $\langle \mathtt{m}:() \to \mathtt{C}\rangle$
- $\langle \mathtt{m}:() \to \langle \mathtt{m}:() \to \mathtt{C}\rangle\rangle$
- $\langle \mathtt{m}:() \to \langle \mathtt{m}:() \to \langle \mathtt{m}:() \to \mathtt{C}\rangle\rangle\rangle$

Roadblock

- The type system is not very *practical*:

```
class C extends Object
{
        C m() {return this;}
}
```

What types does the term `new C()` have?

  - $\mathtt{C}$
  - $\langle \mathtt{m} : () \rightarrow \mathtt{C} \rangle$
  - $\langle \mathtt{m} : () \rightarrow \langle \mathtt{m} : () \rightarrow \mathtt{C} \rangle \rangle$
  - $\langle \mathtt{m} : () \rightarrow \langle \mathtt{m} : () \rightarrow \langle \mathtt{m} : () \rightarrow \mathtt{C} \rangle \rangle \rangle$
  - etc ...

- None of these types are *principal* – we cannot build an algorithm for typing objects based on recursively defined classes.

Solution: Recursive Types?

Recursive types are *finite* representations of *infinite* types.

For example, the type $\mu X.\langle \mathtt{m}\!:\!() \to X\rangle$ represents the infinite type

$$\langle \mathtt{m}\!:\!() \to \langle \mathtt{m}\!:\!() \to \langle \mathtt{m}\!:\!() \to \ldots\rangle\rangle\rangle$$

- This makes it perfect to describe the behaviour of our example object, `new C()`.
- The type $\mu X.\langle \mathtt{m}\!:\!() \to X\rangle$ is 'larger' than all of the types we saw on the previous slide: is it PRINCIPAL.

N.B. A recursive type and its unfolding are considered *equivalent*, so the two representations can be swapped one for another during type assignment.

(We will see this on the next slide)

But Wait ...

- Recursive types (in unrestricted form) are *logically inconsistent*:

$$\mu X.X \to A \simeq (\mu X.X \to A) \to A$$

$$\dfrac{\begin{array}{c}\mathcal{D}\\ \vdash \lambda x.xx{:}(\mu X.X \to A) \to A\end{array} \qquad \dfrac{\begin{array}{c}\mathcal{D}\\ \vdash \lambda x.xx{:}(\mu X.X \to A) \to A\end{array}}{\vdash \lambda x.xx{:}\mu X.X \to A}\,(\simeq)}{\vdash (\lambda x.xx)(\lambda x.xx){:}A}$$

This is Curry's Paradox

- So non-termination is typeable!
  - Not necessarily a problem if you are only interested in *partial* correctness:

    "Typeable programs won't go wrong ... but they may not return a result"
  - But we are looking for a basis for a *fully abstract* analysis!
- Mendler's restriction [1] gives us back termination, but at the expense of natural types for useful OO features like *binary methods*.

Nakano to the Rescue

- Nakano [2] introduces a 'modal' type constructor $\bullet$ which controls the folding of recursive types

$$\bullet(\mu X.\bullet X \to A) \to A \simeq \mu X.\bullet X \to A$$

$$(\mu X.\bullet X \to A) \to A \not\simeq \mu X.\bullet X \to A$$

- No more Curry's Paradox:

$$\dfrac{\begin{array}{c}\mathcal{D}\\ \vdash \lambda x.xx:(\mu X.\bullet X \to A) \to A\end{array} \qquad \dfrac{\begin{array}{c}\mathcal{D}\\ \vdash \lambda x.xx:(\mu X.\bullet X \to A) \to A\end{array}}{\nvdash \lambda x.xx:\mu X.\bullet X \to A}\,(\not\simeq)}{\nvdash (\lambda x.xx)(\lambda x.xx):A}$$

  – Caveat: Nakano's sytem is only *head* normalising - typeable programs may not terminate, but will always [continue to] produce output.

- Can we use a similar trick for typing Featherweight Java?

Detour: Typing Recursive Definitions

Take a (familiar and functional) recursive definition, e.g.

$$\text{add} \equiv \lambda xy.\text{case } x \text{ of } \texttt{Zero} \text{ => } y \mid \texttt{Suc}(n) \text{ => } \texttt{Suc}(\text{add} \;\; n \;\; y)$$

The solution of this equation is a fixed point, so we denote the term satisfying this definition by

$$\text{Fix add}.\lambda xy.\text{case } x \text{ of } \texttt{Zero} \text{ => } y \mid \texttt{Suc}(n) \text{ => } \texttt{Suc}(\text{add} \;\; n \;\; y)$$

And it can be obtained using a *fixed point operator* $\mathbf{Y}$:

$$\text{add} = \mathbf{Y}(\lambda f.\lambda xy.\text{case } x \text{ of } \texttt{Zero} \text{ => } y \mid \texttt{Suc}(n) \text{ => } \texttt{Suc}(f \;\; n \;\; y))$$

Using recursive types, $\mathbf{Y}$ has the type scheme $(A \to A) \to A$; so to type a recursive definition

$$\frac{\vdash \mathbf{Y}{:}(A \to A) \to A \qquad \dfrac{f{:}A \vdash M{:}A}{\vdash \lambda f.M{:}A \to A}}{\vdash \mathbf{Y}(\lambda f.M){:}A} \qquad\qquad \frac{f{:}A \vdash M{:}A}{\vdash \text{Fix } f.M{:}A}$$

Objects are Built from Recursive Definitions

A term representing a recursively defined function reduces like this:

$$
\begin{aligned}
(\mathsf{Fix}\ \mathtt{this}.(\lambda\overrightarrow{x_n}.\mathsf{e}_b))\ \overrightarrow{\mathsf{e}_n} &\rightarrow (\mathsf{e}_b[\mathsf{Fix}\ \mathtt{this}.(\lambda\overrightarrow{x_n}.\mathsf{e}_b)/\mathtt{this}])\ \overrightarrow{\mathsf{e}_n} \\
&\rightarrow \ \mathsf{e}_b[\mathsf{Fix}\ \mathtt{this}.(\lambda\overrightarrow{x_n}.\mathsf{e}_b)/\mathtt{this}, \overrightarrow{\mathsf{e}_n}/\overrightarrow{x_n}]
\end{aligned}
$$

Notice the similarity to method invocation, where the defintion of the method `m` in class C is $\mathtt{m}(\overrightarrow{x_n})\ \{\ \mathtt{return}\ \mathsf{e}_b\ \}$:

$$
\mathtt{new}\ \mathtt{C}(\overrightarrow{\mathsf{e'}}).\mathtt{m}(\overrightarrow{\mathsf{e}_n}) \rightarrow \mathsf{e}_b[\mathtt{new}\ \mathtt{C}(\overrightarrow{\mathsf{e'}})/\mathtt{this}, \overrightarrow{\mathsf{e}_n}/\overrightarrow{x_n}]
$$

So, can we type objects (and method invocations) as follows?

$$
\dfrac{\dfrac{\dfrac{\mathsf{C}:\langle \mathtt{m}:(\overrightarrow{\sigma_n}) \rightarrow \tau\rangle, \overrightarrow{x:\sigma_n} \vdash \mathsf{e}_b:\tau}{\mathsf{C}:\langle \mathtt{m}:(\overrightarrow{\sigma_n}) \rightarrow \tau\rangle \vdash \mathtt{m}(\overrightarrow{x_n})\ \{\ \mathtt{return}\ \mathsf{e}_b\ \}:\langle \mathtt{m}:(\overrightarrow{\sigma_n}) \rightarrow \tau\rangle}}{\vdash \mathtt{new}\ \mathtt{C}(\overrightarrow{\mathsf{e'}}):\langle \mathtt{m}:(\overrightarrow{\sigma_n}) \rightarrow \tau\rangle} \qquad \vdash \mathsf{e}_1:\sigma_1 \ldots \vdash \mathsf{e}_n:\sigma_n}{\vdash \mathtt{new}\ \mathtt{C}(\overrightarrow{\mathsf{e'}}).\mathtt{m}(\overrightarrow{\mathsf{e}_n}):\tau}
$$

Putting It All Together

In Nakano's system, fixed point operators have the type scheme $(\bullet A \to A) \to A$, so our typing rule for objects becomes:

$$\frac{\mathtt{C}{:}\bullet\langle \mathtt{m}{:}(\overrightarrow{\sigma_n}) \to \tau\rangle, \overrightarrow{x{:}\sigma_n} \vdash \mathtt{e}_b{:}\tau}{\vdash \mathtt{new}\ \mathtt{C}(\overrightarrow{\mathtt{e'}}){:}\langle \mathtt{m}{:}(\overrightarrow{\sigma_n}) \to \tau\rangle}$$

We can now give our `new C()` example its natural recursive type:

$$\frac{\overline{\mathtt{C}{:}\bullet\mu X.\langle \mathtt{m}{:}() \to \bullet X\rangle \vdash \mathtt{this}{:}\bullet\mu X.\langle \mathtt{m}{:}() \to \bullet X\rangle}}{\vdash \mathtt{new}\ \mathtt{C}(){:}\mu X.\langle \mathtt{m}{:}() \to \bullet X\rangle}$$

Notice that $\mu X.\langle \mathtt{m}{:}() \to \bullet X\rangle$ is really the infinite type $\langle \mathtt{m}{:}() \to \bullet\langle \mathtt{m}{:}() \to \bullet \ldots\rangle\rangle$, and that the result type

$$\bullet\langle \mathtt{m}{:}() \to \bullet \ldots\rangle = \bullet\mu X.\langle \mathtt{m}{:}() \to \bullet X\rangle = \bullet X[\mu X.\langle \mathtt{m}{:}() \to \bullet X\rangle / X]$$

is exactly the type that we have assigned to the body of the `m` method.

The Catch ...

As in Nakano's system, some non-termination slips through the net; consider:

```
class Y extends App {
        App app(App x) {return x.app(this.app(x));}
}
```

The term `new Y().app(z)` is non-terminating:

$$
\begin{aligned}
\texttt{new Y().app(z)} &\rightarrow \texttt{z.app(new Y().app(z))} \\
&\rightarrow \texttt{z.app(z.app(new Y().app(z)))}
\end{aligned}
$$

The Catch ...

As in Nakano's system, some non-termination slips through the net; consider:

```
class Y extends App {
      App app(App x) {return x.app(this.app(x));}
}
```

The term `new Y().app(z)` is non-terminating, but typeable:

$$
\dfrac{
  \dfrac{
    \dfrac{}{\texttt{Y}{:}\bullet\sigma,\texttt{x}{:}\tau \vdash \texttt{x}{:}\tau}
    \qquad
    \dfrac{
      \dfrac{}{\texttt{Y}{:}\bullet\sigma,\texttt{x}{:}\tau \vdash \texttt{this}{:}\bullet\sigma}
      \quad
      \dfrac{\dfrac{}{\texttt{Y}{:}\bullet\sigma,\texttt{x}{:}\tau \vdash \texttt{x}{:}\tau}}{\texttt{Y}{:}\bullet\sigma,\texttt{x}{:}\tau \vdash \texttt{x}{:}\bullet\tau}
    }{\texttt{Y}{:}\bullet\sigma,\texttt{x}{:}\tau \vdash \texttt{this.app(x)}{:}\bullet\bullet\sigma}
  }{
    \dfrac{\texttt{Y}{:}\bullet\sigma,\texttt{x}{:}\tau \vdash \texttt{x.app(this.app(x))}{:}\bullet\sigma}{\texttt{z}{:}\tau \vdash \texttt{new Y()}{:}\sigma}
  }
  \qquad
  \dfrac{}{\texttt{z}{:}\tau \vdash \texttt{z}{:}\tau}
}{\texttt{z}{:}\tau \vdash \texttt{new Y().app(z)}{:}\bullet\sigma}
$$

$$\sigma = \mu X.\langle \texttt{app} : (\mu Y.\langle \texttt{app} : (\bullet\bullet X) \to \bullet X\rangle) \to \bullet X\rangle$$

$$\tau = \mu Y.\langle \texttt{app} : (\bullet\bullet\sigma) \to \bullet\sigma\rangle$$

Extending the Approach

To prevent these non-terminating recursive calls we introduce an additional (modal) type constructor $\circ$, which prevents the *unfolding* of recursive types, thus preventing method invocation.

- We now use this new operator to type self-references (i.e. `this`)

$$
\dfrac{
  \dfrac{}{\mathtt{Y}{:}\circ\sigma, \mathtt{x}{:}\tau \vdash \mathtt{x}{:}\tau}
  \qquad
  \dfrac{
    \dfrac{}{\mathtt{Y}{:}\circ\sigma, \mathtt{x}{:}\tau \vdash \mathtt{this}{:}\circ\sigma}
    \quad
    \dfrac{\dfrac{}{\mathtt{Y}{:}\circ\sigma, \mathtt{x}{:}\tau \vdash \mathtt{x}{:}\tau}}{\mathtt{Y}{:}\circ\sigma, \mathtt{x}{:}\tau \vdash \mathtt{x}{:}\bullet\tau}
  }{\mathtt{Y}{:}\circ\sigma, \mathtt{x}{:}\tau \nvdash \mathtt{this.app(x)}{:}\bullet\bullet\sigma}
}{
  \dfrac{\mathtt{Y}{:}\bullet\sigma, \mathtt{x}{:}\tau \nvdash \mathtt{x.app(this.app(x))}{:}\bullet\sigma}{\nvdash \mathtt{new\ Y()}{:}\sigma}
}
$$

$$\sigma = \mu X.\langle \mathtt{app} : (\mu Y.\langle \mathtt{app} : (\bullet\bullet X) \to \bullet X\rangle) \to \bullet X\rangle$$
$$\tau = \mu Y.\langle \mathtt{app} : (\bullet\bullet\sigma) \to \bullet\sigma\rangle$$

Conclusions

- We have shown how to apply Nakano's approach to FJ;
- We have extended the approach with the $\circ$ type constructor;
- We have shown we can assign recursive types to some *'safe'* examples;
- We have shown the untypeability of some non-terminating, *'unsafe'* examples.

Just a proof-of-concept at this stage – no formal results yet!

References

[1] N.P. Mendler. Recursive Types and Type Constraints in Second Order Lambda Calculus. LICS'87.

[2] H. Nakano. A Modality for Recursion. LICS'00.

[3] R. Rowe and S. van Bakel. Approximation Semantics and Expressive Predicate Assignment for Object-Oriented Programming. TLCA'11.