Safe, Flexible Recursive Types for Featherweight Java

Reuben N. S. Rowe

Imperial College London

Abstract. This paper presents a type assignment system with recursive types for Featherweight Java, inspired by the work of Nakano. Nakano's innovation consists in adding a modal type constructor which acts to control the folding of recursive types, resulting in a head-normalisation guarantee. We build on this approach by introducing a second modal type constructor which prevents the unfolding of types in contexts where doing so results in non-termination. Moreover our system inherits the flexibility of Nakano's approach, allowing object-oriented features (such as binary methods) to be typed in a safe and intuitive way. The work described in this paper is preliminary, and no formal results are claimed. However, we conjecture that our type system enjoys strong normalisation and we motivate this by working through some apposite examples.

1 Introduction

Recursive types can be viewed as finite representations of infinite (but *regular*) types [8, Chapter 20]. For example, the recursive type $T = \mu X.(A \to X)$ represents the infinite type satisfying the (recursive) equation $T = A \to T$. Alternatively, T can be understood to be the type obtained from 'unfolding' $\mu X.(A \to X)$ to $A \to (\mu X.(A \to X))$ an infinite number of times. The folded and unfolded form, denoting the same (infinite) type, are considered to be equivalent, and it is usual to freely exchange one for another during type assignment.

These types naturally capture the behaviour of entities which are (potentially) infinite, or structures of indeterminate size such as lists or streams. Among such entities are the *objects* of object-oriented (OO) programming systems. For example, consider objects which are instances of the following Java classes:

```
class C {
      C m() { return new C(); }
}

class Suc implements Nat {
      Nat pred;
      Nat add(Nat x) { return new Suc(this.pred.add(x)); }
}
```

In the first example, instances of class `C` have a method `m` which returns another instance of `C`; thus given a `C` object, `m` may be safely invoked any arbitrary (and indeterminate) number of times. A natural (recursive) type describing this behaviour could be, for instance, $\mu X.\langle \texttt{m}{:}() \to X\rangle$. The second example gives a class containing a method `add`, representing addition on (positive) natural numbers. The `add` method creates a new `Suc` object and thus, as in the previous example, it may be invoked any arbitrary number of times.

Recursive types, then, provide an ideal mechanism for reasoning about object-oriented programs. Indeed, much work (see e.g. [3, 1, 4, 2]) has already been done on the type-theoretic relationship between recursive types and OO. The drawback to recursive types however is that in their unrestricted form they are *logically inconsistent* - that is to say, they allow for the typing of non-convergent (non-terminating) programs. This is not always a problem from a program analysis point-of-view, provided one is only interested in ensuring the *partial* correctness of programs, but poses problems when constructing type-based semantics, and also when reasoning about termination properties.

It is known that placing syntactic restrictions on recursive types – specifically, disallowing *negative* occurrences of recursively bound type variables (as in the type $\mu X.X \to A$) – restores convergence and logical consistency [6]. However, this approach poses a unique problem within the setting of OO. The `Suc` example illustrates another feature of object-oriented programming: *binary methods*. These are methods which take an argument of the same kind as the object containing it - in the case of our example, the `add` method (belonging to `Suc` objects) takes another `Suc` object as input (besides also returning one as output). This is exactly the behaviour captured by recursive types containing negative self references, and one might expect to be able to assign a type such as $\mu X.\langle \texttt{m}{:}X \to X\rangle$ to instances of `Suc`. Thus, such a restriction on types is unsatisfactory for object-oriented programming.

Nakano has developed a system of recursive types for the λ-calculus, which goes some way to addressing these issues [7]. In his system, there is no restriction on the (negative) occurrence of recursively bound type variables, and by introducing an additional type constructor $\bullet$, a convergent system (up to head-normalisation) is obtained. In previous work [9] the author studied semantics for object-oriented programming based upon intersection types. The current work is motivated by a failure of that work to provide fully decidable type inference in the presence of recursively defined classes, such as those given in the example above.

In this paper, we describe a variation on Nakano's theme which we believe is capable of providing a logically consistent and flexible foundation for OO type theory. We chose to first focus on a system without intersections for simplicity – such a system is easier to formulate and reason about (both formally and informally), and since type systems without intersections are simpy special cases of systems with intersections, the recursive- and intersection-based aspects of the system can be dealt with orthogonally. We point out that the work is at a preliminary stage, so we have no formal results to present. Rather, the aim

is simply to give evidence in support of our thesis in the form of typed (non) examples. Due to space restrictions, we are unable to provide a comprehensive explanation of the relevant background material, so we will assume that the reader is familiar with the basics of type theory and type assignment, and the author's previous cited work.

2 FJ$\circ\mu$: Safe Recursive Types for OO

In this section we describe how we apply Nakano's approach [7] to OO. The system we describe is a variation on our previous work [9], which in turn is based on Featherweight Java (FJ) [5], a formal model of the core operational semantics of Java. We refer the reader to those papers for details elided here.

Definition 1 (FJ$\circ\mu$ Predicates). The predicates (types) of FJ$\circ\mu$ are formed according to the following grammar (where X, like φ, ranges over predicate variables):

$$\sigma \quad ::= \quad C \quad | \quad \varphi \quad | \quad \bullet\,\sigma \quad | \quad \circ\,\sigma \quad | \quad \langle f{:}\sigma\rangle \quad | \quad \mu X.\langle m{:}(\overrightarrow{\sigma_n}) \to \sigma\rangle$$

with the restriction that any bound recursive predicate variables must be within the scope of either the $\bullet$ or $\circ$ type constructors (under their respective μ binders). The notation $\bullet^r\,\sigma$ ($\circ^r\,\sigma$) is a shorthand for σ preceded by r occurrences of $\bullet$ ($\circ$), and $\bullet^{r+}\,\sigma$ ($\circ^{r+}\,\sigma$) denotes the same thing, but indicates a strictly positive number of occurrences of $\bullet$ or $\circ$.

The basic idea is that method invocations are allowed by types of the form $\bullet^r\,\mu X.\langle m{:}(\vec{\sigma}) \to \sigma\rangle$, but *disallowed* at types of the form $\circ^{r+}\,\mu X.\langle m{:}(\vec{\sigma}) \to \sigma\rangle$. Thus the $\circ$ constructor serves to control the unfolding of recursive types, and can therefore be seen in some respect as the *dual*[1] of the $\bullet$ constructor in Nakano's system where it is used to control the *folding* of recursive types.

We define a *coercion* relation on predicates which permits $\bullet$ types to turn into $\circ$ types, and is used to determine when a method predicate can be safely assigned to a new object instance.

Definition 2 (Coercion). The *coercion* relation $\lhd$ is the smallest preorder on predicates satisfying:

$$\sigma \lhd \sigma' \Rightarrow \begin{cases} \bullet\,\sigma \lhd \bullet\,\sigma' \\ \bullet\,\sigma \lhd \circ\,\sigma' \\ \circ\,\sigma \lhd \circ\,\sigma' \end{cases}$$

The type assignment system for FJ$\circ\mu$ is given by the rules in Figure 1. Γ is a typing environment for variables, and Σ is a typing environment for *classes* (also called a *self* environment), used to type `new` expressions and the self reference variable `this` within the bodies of methods. These class environments contain a

[1] Here we use this word in an informal sense, rather than its formal category-theoretic sense.

$$(\textsc{var}):\ \frac{}{\Sigma;\Gamma,x{:}\sigma\vdash x{:}\sigma}\ (x\neq\texttt{this})\qquad (\bullet):\ \frac{\Sigma;\Gamma\vdash e{:}\sigma}{\Sigma;\Gamma\vdash e{:}\bullet\,\sigma}\qquad (\textsc{coerce}):\ \frac{\Sigma;\Gamma\vdash e{:}\sigma}{\Sigma;\Gamma\vdash e{:}\sigma'}\ (\sigma\lhd\sigma')$$

$$(\textsc{fld}):\ \frac{\Sigma;\Gamma\vdash e{:}\bullet^r\langle f{:}\sigma\rangle}{\Sigma;\Gamma\vdash e\,.\,f{:}\bullet^r\,\sigma}\qquad (\textsc{invk}):\ \frac{\Sigma;\Gamma\vdash e_0{:}\bullet^r\,\mu X.\langle m{:}(\overrightarrow{\sigma_n})\to\sigma\rangle\quad \Sigma;\Gamma\vdash e_1{:}\bullet^r\,\sigma'_1\quad\ldots\quad\Sigma;\Gamma\vdash e_n{:}\bullet^r\,\sigma'_n}{\Sigma;\Gamma\vdash e_0\,.\,m(\overrightarrow{e_n}){:}\bullet^r(\sigma[\mu X.\langle m{:}(\overrightarrow{\sigma_n})\to\sigma\rangle/X])\quad(\sigma'_i=\sigma_i[\mu X.\langle m{:}(\overrightarrow{\sigma_n})\to\sigma\rangle/X]\text{ for each }i\in\overline{n})}$$

$$(\textsc{self-obj}):\ \frac{}{\Sigma,\widehat{C}{:}(\overrightarrow{\sigma_n})\to\sigma;\Gamma\vdash\texttt{this}{:}C}\qquad (\textsc{self-fld}):\ \frac{}{\Sigma,\widehat{C}{:}(\overrightarrow{\sigma_n})\to\sigma;\Gamma\vdash\texttt{this}{:}\langle f_i{:}\sigma_i\rangle}\ (\mathcal{F}(C)=\overrightarrow{f_n},\ i\in\overline{n})$$

$$(\textsc{self-meth}):\ \frac{}{\Sigma,\widehat{C}{:}(\overrightarrow{\sigma_n})\to\sigma;\Gamma\vdash\texttt{this}{:}\circ\,\sigma}\qquad (\textsc{inst-obj}):\ \frac{\Sigma;\Gamma\vdash e_1{:}\sigma_1\quad\ldots\quad\Sigma;\Gamma\vdash e_n{:}\sigma_n}{\Sigma;\Gamma\vdash\texttt{new}\ C(\overrightarrow{e_n}){:}C}\ (\mathcal{F}(C)=\overrightarrow{f_n})$$

$$(\textsc{inst-fld}_1):\ \frac{\ldots\quad\Sigma;\Gamma\vdash e_i{:}\bullet^r\,\sigma\quad\ldots}{\Sigma;\Gamma\vdash\texttt{new}\ C(\overrightarrow{e_n}){:}\bullet^r\langle f_i{:}\sigma\rangle}\ (\mathcal{F}(C)=\overrightarrow{f_n},\ i\in\overline{n})\qquad (\textsc{inst-fld}_2):\ \frac{\ldots\quad\Sigma;\Gamma\vdash e_i{:}\circ^r\,\sigma\quad\ldots}{\Sigma;\Gamma\vdash\texttt{new}\ C(\overrightarrow{e_n}){:}\circ^r\langle f_i{:}\sigma\rangle}\ (\mathcal{F}(C)=\overrightarrow{f_n},\ i\in\overline{n})$$

$$(\textsc{inst-rec}):\ \frac{\Sigma;\Gamma\vdash e_1{:}\circ^{r+}\sigma_1\quad\ldots\quad\Sigma;\Gamma\vdash e_n{:}\circ^{r+}\sigma_n}{\Sigma;\Gamma\vdash\texttt{new}\ C(\overrightarrow{e_n}){:}\circ^{r+}\sigma}\ (C{:}(\overrightarrow{\sigma_n})\to\sigma\in\Sigma)$$

$$(\textsc{inst-meth}_1):\ \frac{\overline{\Sigma},\widehat{C}{:}(\overrightarrow{\sigma_n})\to\mu X.\langle m{:}(\overrightarrow{\sigma'_{n'}})\to\sigma'\rangle;x_1{:}\sigma''_1,\ldots,x_{n'}{:}\sigma''_{n'}\vdash e_b{:}\sigma''\quad\Sigma;\Gamma\vdash e_i{:}\bullet^r\,\sigma_i\ (\forall i\in\overline{n})}{\Sigma;\Gamma\vdash\texttt{new}\ C(\overrightarrow{e_n}){:}\bullet^r\,\mu X.\langle m{:}(\overrightarrow{\sigma'_{n'}})\to\sigma'\rangle}\ (*)$$

$$(\textsc{inst-meth}_2):\ \frac{\overline{\Sigma},\widehat{C}{:}(\overrightarrow{\sigma_n})\to\mu X.\langle m{:}(\overrightarrow{\sigma'_{n'}})\to\sigma'\rangle;x_1{:}\sigma''_1,\ldots,x_{n'}{:}\sigma''_{n'}\vdash e_b{:}\sigma''\quad\Sigma;\Gamma\vdash e_i{:}\circ^{r+}\sigma_i\ (\forall i\in\overline{n})}{\Sigma;\Gamma\vdash\texttt{new}\ C(\overrightarrow{e_n}){:}\circ^{r+}\mu X.\langle m{:}(\overrightarrow{\sigma'_{n'}})\to\sigma'\rangle}\ (*)$$

$$*\quad(\mathcal{M}(C,m)=(\overrightarrow{x_{n'}},e_b),\sigma'[\mu X.\langle m{:}(\overrightarrow{\sigma'_{n'}})\to\sigma'\rangle/X]\lhd\sigma'',\sigma''_i=\sigma'_i[\mu X.\langle m{:}(\overrightarrow{\sigma'_{n'}})\to\sigma'\rangle/X]\text{ for each }i\in\overline{n'})$$

Fig. 1. Predicate Assignment for FJ$\circ\mu$

unique *marked* class, indicated by $\widehat{C}$ and used to keep track of which class the method body currently being typed appears in. The notation $\overline{\Sigma}$ represents the self environment identical to Σ, except that no class is marked. Valid environments may only contain a *single* type statement for each variable or class. The notation $\sigma_1[\sigma_2/X]$ stands for the type obtained from σ_1 by replacing all (free) occurrences of X with σ_2.

The key inference rules of the type system are the two (INST-METH) rules, which assign a (recursive) *method* predicate to a new object (instance). Their operation can be understood by viewing the `new` keyword as representing a function that constructs objects from class definitions. Since classes may themselves create new objects according to their own definition (i.e. call their own `new` function), these functions are recursively defined. Thus the rule takes the familiar form for typing a recursively defined term, in which the body of the term is typed using an environment where recursive calls must be typed with the *same* type as the body itself. There is a subtle twist however - since the type scheme for fixed point operators in $\lambda\bullet\mu$ is $(\bullet\,A\to A)\to A$, recursively created objects must now be typed not with $\mu X.\langle m{:}(\overrightarrow{\sigma_n})\to\sigma\rangle$, but with a *bulleted* version of this type. Nakano's approach would suggest using a $\bullet$ type, however in our system this would permit recursive method invocations resulting in non-termination, and so

instead we use the type $\circ\,\mu X.\langle m{:}(\overrightarrow{\sigma_n}) \to \sigma\rangle$, preventing such invocations. This is enforced by the $\circ^{r+}$ in the (INST-REC) and (SELF-METH) rules.

Using this type system, the examples from the introduction can be given their expected types:

$$\dfrac{\dfrac{}{\widehat{\texttt{C}}{:}() \to \mu X.\langle \texttt{m}{:}() \to \bullet\, X\rangle \vdash \texttt{new C()}{:}\circ\,\mu X.\langle \texttt{m}{:}() \to \bullet\, X\rangle}\,(\textsc{inst-rec})}{\vdash \texttt{new C()}{:}\mu X.\langle \texttt{m}{:}() \to \bullet\, X\rangle}\,(\textsc{inst-meth}_1)$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\widehat{\texttt{Suc}}{:}(\sigma) \to \sigma; \texttt{x}{:}\bullet\,\sigma \vdash \texttt{this}{:}\langle \texttt{pred}{:}\sigma\rangle}\,(\textsc{self-fld})}{\widehat{\texttt{Suc}}{:}(\sigma) \to S; \texttt{x}{:}\bullet\,\sigma \vdash \texttt{this.pred}{:}\sigma}\,(\textsc{fld}) \quad \dfrac{}{\widehat{\texttt{Suc}}{:}(\sigma) \to \sigma; \texttt{x}{:}\bullet\,\sigma \vdash \texttt{x}{:}\bullet\,\sigma}\,(\textsc{var})}{\widehat{\texttt{Suc}}{:}(\sigma) \to \sigma; \texttt{x}{:}\bullet\,\sigma \vdash \texttt{this.pred.add(x)}{:}\bullet\,\sigma}\,(\textsc{invk})}{\widehat{\texttt{Suc}}{:}(\sigma) \to \sigma; \texttt{x}{:}\bullet\,\sigma \vdash \texttt{this.pred.add(x)}{:}\circ\,\sigma}\,(\textsc{coerce})}{\widehat{\texttt{Suc}}{:}(\sigma) \to \sigma; \texttt{x}{:}\bullet\,\sigma \vdash \texttt{new Suc(this.pred.add(x))}{:}\circ\,\sigma}\,(\textsc{inst-rec}) \quad \dfrac{}{\texttt{y}{:}\sigma \vdash \texttt{y}{:}\sigma}\,(\textsc{var})}{\texttt{y}{:}\sigma \vdash \texttt{new Suc(y)}{:}\sigma}\,(\textsc{inst-meth}_1)$$

where $\sigma = \mu X.\langle \texttt{add}{:}(\bullet\, X) \to \bullet\, X\rangle$.

It also prevents the typing of non-terminating programs. Consider the following classes (where the `App` interface declares the method `app`):

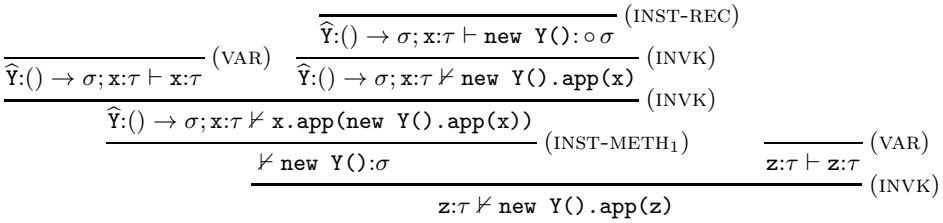
```
class D { D m() { return new D().m(); } }

class Y implements App {
      App app(App x) { return x.app(new Y().app(x)); }
}
```

The methods in both these classes make recursive calls leading to non-terminating behaviour: the expression `new D().m()` is *unsolvable*, as it reduces only to itself; and the method invocation `new Y().app(z)`, although it reduces in one step to a *head* normal form, has the infinite reduction sequence:

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

Both of these expressions are *untypable* in FJ$\circ\mu$, since the presence of the $\circ$ type constructor prevents the typing of the recursive method invocations which lead to the non-termination.

$$\dfrac{\dfrac{\dfrac{}{\widehat{\texttt{D}}{:}() \to \mu X.\langle \texttt{m}{:}() \to \varphi\rangle \vdash \texttt{new D()}{:}\circ\,\mu X.\langle \texttt{m}{:}() \to \varphi\rangle}\,(\textsc{inst-rec})}{\widehat{\texttt{D}}{:}() \to \mu X.\langle \texttt{m}{:}() \to \varphi\rangle \nvdash \texttt{new D().m()}}\,(\textsc{invk})}{\nvdash \texttt{new D()}{:}\mu X.\langle \texttt{m}{:}() \to \varphi\rangle}\,(\textsc{inst-meth}_1)}{\nvdash \texttt{new D().m()}}\,(\textsc{invk})$$



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

3 Conclusions

We have presented a type system for a variant of Featherweight Java which assigns recursive types to class-based object-oriented programs. It is inspired by previous work on head normalising recursive types for Lambda Calculus, and we give several examples which show that our type system (a) types (persistently) normalising terms with intuitive recursive types; and (b) does *not* type non-terminating programs. Our contribution consists in showing how Nakano's approach can be applied to OO and, more importantly, in its extension in the form of the second type constructor $\circ$. The latter in particular is a novel contribution of this paper. We conjecture that our system types only strongly normalising (i.e. terminating) terms, and proving this is an important task of future research.

A principal type for a term is a type from which all other types assignable to that term can be generated. If a type assignment system has the principal type property, then a principal type exists for all typeable terms. Such a property is the key component of any algorithm for deciding type assignment, and thus typeability. Our motivation in carrying out this research was to obtain a type system for Featherweight Java of the same flavour as our previous work, but with *finite* (and thus *decidable*) principal types for objects. We feel the system presented in this paper is a good candidate. Take our first example from the introduction: if the set of *principal* types for this program in FJ$\circ\mu$ is $\{\mathtt{C}, \mu X.\langle \mathtt{m}{:}() \to \bullet X\rangle\}$, then by 'unfolding' this set in a similar way to that described in the introduction (i.e. by replacing the recursive components, $\bullet X$, by other types in the set) we obtain the infinite set $\{\mathtt{C}, \langle \mathtt{m}{:}() \to \mathtt{C}\rangle, \langle \mathtt{m}{:}() \to \langle \mathtt{m}{:}() \to \mathtt{C}\rangle\rangle, \ldots\}$, which is the set of principal types for this example in our system without recursive types.

The next step of future research, after showing normalisation and (finite) principal typings for this system, will be to add *intersections* to the type system as in our previous work in order to build a fully abstract semantics based on our recursive types. It is our ultimate aim to then construct decidable type inference systems for this augmented type assignment resulting in expressive, powerful and practical type based analysis of class-based OO programs.

References

1. M. Abadi and L. Cardelli. *A Theory Of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.
2. C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for javascript. In *ECOOP*, pages 428–452, 2005.
3. K. Fisher, F. Honsell, and J. C. Mitchell. A Lambda Calculus of Objects and Method Specialization. *Nordic J. of Computing*, 1(1):3–37, 1994.
4. N. Glew. An Efficient Class and Object Encoding. In *Proceedings of OOPSLA00*, pages 311–324, 2000.
5. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. In *OOPSLA*, pages 132–146, 1999.
6. N.P. Mendler. Recursive Types and Type Constraints in Second Order Lambda Calculus. In *Proceedings of LICS'87*, pages 30–36. IEEE, 1987.
7. H. Nakano. A Modality for Recursion. In *LICS*, pages 255–266, 2000.
8. Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
9. R. Rowe and S. van Bakel. Approximation Semantics and Expressive Predicate Assignment for Object-Oriented Programming. In *TLCA*, pages 229–244, 2011.