Modelling Java Requires State

Alexander J. Summers
Imperial College London
180 Queens Gate
South Kensington, London
alexander.j.summers@imperial.ac.uk

ABSTRACT

Interesting questions concerning Java-like languages are often studied in the context of smaller programming calculi such as Featherweight Java. The simplicity of the syntax, and small number of features, and in particular the lack of state, make it possible to focus on the issues of interest. Although the programming languages are imperative, Featherweight Java and various similar calculi are functional.

We argue that the study of the type system of Java 5.0 and beyond requires a calculus with state. For example, the treatment of wildcards in Java is tailored to preserve soundness in the presence of stateful computation, a feature that is not present in functional calculi. A stateful calculus is necessary before the potential pitfalls of an incorrect proposal can be seen. We illustrate this point by showing that a traditional treatment of existential types (based on that historically known for the Lambda Calculus) is unsound for Java but remains sound for Featherweight Java.

1. BACKGROUND

1.1 Featherweight Java

Featherweight Java and Featherweight Generic Java [1] are functional (expression-based) calculi, intended to provide a minimal basis on which to study features and extensions of Java. For the purposes of this paper, it will suffice to consider only the syntax of *expressions* in Featherweight Generic Java (hereafter FGJ) - the aspects of the calculus concerned with class and method declaration will be elided in this paper. A slight variation of the expression syntax in FGJ is given in the following definition. We omit casts and generic methods for brevity.

DEFINITION 1 (FGJ EXPRESSIONS [1]). *Class types N are defined below. We use the vector notation $\overrightarrow{N_i}$ to denote a sequence (of class types) indexed by i.*

$$N := C\!<\!\overrightarrow{N_i}\!>$$

Expressions, ranged over by d and e, are defined over an infinite set of variables, ranged over by x and y, field names ranged over by f and g, and method names ranged over by m and n.

$$d, e \;:=\; x \mid e.f \mid e.m(\overrightarrow{e_i}) \mid \mathsf{new}\ N(\overrightarrow{e_i})$$

Values, indexed by u and v are defined by the following syntactic subset:

$$u, v \;:=\; \mathsf{new}\ N(\overrightarrow{v_i})$$

The calculus includes variables (which are place-holders for method parameters, and the special variable this), field lookup, method call and object creation. Note that there is no assignment in the calculus; FGJ is purely functional.

We write $e_1\{e_2/x\}$ for the capture-avoiding subsitution of e_2 for x in e_1 (and use similar syntax for capture-avoiding substitutions on type variables later in the paper).

Since we elide the declarations of classes and methods in this paper, we assume the existence of three functions: *fields* (to retrieve all of the (typed) fields declared in a class), *mbody* to retrieve the argument names and method body of a method in a class, and *mType* to retrieve a type signature $\overrightarrow{T_i}\!\to\!T$. We refer the reader to the FJ paper [1] for details.

Contrary to the usual presentations of the calculus, we present the reductions with a large-step semantics, since this avoids some technical difficulties with the interaction between existential types and small-step reduction rules. We present the call-by-value version of the calculus, based on the presentation of Pierce [3], rather than the unrestricted system of the original work [1]. The reduction relation $e \to v$ for full reduction of expressions e to values v is defined as follows:

DEFINITION 2 (FGJ BIG-STEP SEMANTICS).

$$\frac{\begin{array}{l} e \to \mathsf{new}\ N(\overrightarrow{v_i}) \\ \mathit{fields}(N) = (\overrightarrow{T_i\ f_i}) \\ e_j \to v \end{array}}{e.f_j \to v}\ (\mathit{field})$$

$$\frac{\begin{array}{l} e \to \mathsf{new}\ N(\overrightarrow{v_i}) \\ \mathit{mBody}(m, N) = (\overrightarrow{x_j}, e_b) \\ \overrightarrow{e_j \to v_j} \\ e_b\{(\mathsf{new}\ N(\overrightarrow{v_i}))/\mathsf{this}\}\{\overrightarrow{v_j/x_j}\} \to v \end{array}}{e.m(\overrightarrow{e_j}) \to v}\ (\mathit{method})$$

Note that reduction is only defined for closed expressions.

FTfJP '09, July 6 2009, Genova, Italy

$$\frac{x:T \in \Gamma}{\Gamma \vdash_\exists x:T}\ (\text{T-VAR}) \qquad \frac{\Gamma \vdash_\exists e:C\overrightarrow{<T_i>} \quad \mathit{fields}(C\overrightarrow{<T_i>}) = (\overrightarrow{T_j\ f_j})}{\Gamma \vdash_\exists e.f_k:T_k}\ (\text{T-FIELD})$$

$$\frac{\Gamma \vdash_\exists e:C\overrightarrow{<T_i>} \quad \mathit{mType}(m, C\overrightarrow{<T_i>}) = (\overrightarrow{T_j} \to T) \quad \overrightarrow{\Gamma \vdash_\exists e_j:T'_j} \quad \overrightarrow{T'_j \preceq T_j}}{\Gamma \vdash_\exists e.m(\overrightarrow{e_j}):T}\ (\text{T-INVK})$$

$$\frac{\mathit{fields}(C\overrightarrow{<T_i>}) = (\overrightarrow{T_j\ f_j}) \quad \overrightarrow{\Gamma \vdash_\exists e_j:T'_j} \quad \overrightarrow{T'_j \preceq T_j}}{\Gamma \vdash_\exists \mathsf{new}\ C\overrightarrow{<T_i>}(\overrightarrow{e_j}):C\overrightarrow{<T_i>}}\ (\text{T-NEW})$$

$$\frac{\Gamma \vdash_\exists e:T\{T'/X\}}{\Gamma \vdash_\exists e:\exists X.T}\ (\exists\mathcal{I}) \qquad \frac{\Gamma \vdash_\exists e_1:\exists X.T \quad \Gamma, y:T\{Y/X\} \vdash_\exists e_2:T'}{\Gamma \vdash_\exists e_2\{e_1/y\}:T'}\ (\exists\mathcal{E})^*$$

* if $Y \notin \Gamma$ and $Y \notin T'$.

Figure 1: Type Assignment with Existential Types

1.2 Existential Types

The classical work on existential types was developed in the context of the λ-calculus, which forms the basis of the functional programming paradigm. In this context, existential types are used to model data abstraction, in the sense of modules which can hide details of their implementations from clients.

It is well-known that existential types can be used to understand the wildcards of the Java type system [4]. A Java type which includes wildcards can be understood as an existential type by replacing each wildcard type of the form $C<?>$ by an existential type of the form $\exists X.C<X>$. For example, the Java type $C<?, D<?>>$ can be understood as a shorthand for the existential type $\exists X.C<X, \exists Y.D<Y>>$. For simplicity, in this paper we will only consider unbounded existential types (and hence, unbounded wildcards), since the conclusions of the paper are orthogonal to the treatment of bounds.

The traditional approach to existential types [2] (which is based on a typed calculus) includes introducing explicit term syntax corresponding with both the introduction and elimination of existential types. However, in the setting of an untyped call-by-value calculus this presentation can be simplified: it suffices simply to add the logical rules for existential introduction and elimination to the type system, without changing the syntax. We consider taking exactly this approach for FGJ, by adding the following two type-assignment rules:

Definition 3 (Existential Types Rules).

$$\frac{\Gamma \vdash e:T\{T'/X\}}{\Gamma \vdash e:\exists X.T}\ (\exists\mathcal{I})$$

$$\frac{\Gamma \vdash e_1:\exists X.T \quad \Gamma, y:T\{Y/X\} \vdash e_2:T'}{\Gamma \vdash e_2\{e_1/y\}:T'}\ (\exists\mathcal{E})^*$$

* *if* $Y \notin \Gamma$ *and* $Y \notin T'$.

As is typical for existential types, the introduction rule allows the use of a new existentially-bound variable to "hide" a type T' called the *witness* type of the existential type. The elimination rule is more complex, but can be understood as follows: e_1 has an existentially-quantified type in which a witness type is hidden. We cannot know exactly what this type is, so if we want to make use of it temporarily, we should represent it with a fresh type variable Y (the side-condition on the rule formalises this notion of "freshness"). However, Y is only a hypothetical type variable, which does not have a meaning in the original scope of the rule - therefore Y can only be used in the sub-derivation for typing e_2, and may not escape in the conclusion of the rule. Computationally, the implicit argument here is that an expression of existential type will still contain a value of some non-quantified (but unknown) type, and if this partial information is enough to deduce well-typedness of a further expression in which it is used, the rule will make this possible.

This treatment of existential types is rather different to the treatment of typing wildcards in Java. In the case of Java, essentially every time an expression of existential type is subject to field or method lookup, a fresh name for the unknown witness type is generated. This could be implemented using the rules above by restricting the $(\exists\mathcal{E})$ rule to only allow expressions e_2 of the form $y.f$ and $y.m(\overrightarrow{e_i})$ in which y does not occur free in the expressions $\overrightarrow{e_i}$. This is a true restriction, in the sense that many fewer expressions become typeable for a given context Γ. This restriction seems unmotivated from the point of view of existential types, since it breaks the correspondence with the usual logical rules. However, it does rule out the potential unsoundness in the type system which we illustrate in this paper. Our point is that the proposal we outline here could easily have been (and perhaps was) considered as an alternative to the notion of wildcards which was added to Java, but its flaws could not have been exposed in the context of a functional calculus - the dangers are only apparent in the presence of state.

2. EXISTENTIAL TYPES FOR FGJ

Let us consider the type system for FGJ augmented with the unrestricted rules of Definition 3. We write $\Gamma \vdash_\exists e:T$ for typing judgements in this extended system:

Definition 4 (Existential Types for FGJ). *Types are defined as follows :*

$$T \ ::= \ X \ \mid \ C\overrightarrow{<T_i>} \ \mid \ \exists X.T$$

We elide questions of well-formedness for types (e.g., concerning the treatment of free type variables), since solutions to these problems are standard and not particularly relevant

$$\frac{\Gamma \vdash_\exists e_1 : C\overrightarrow{<T_i>} \quad \mathit{fields}(C\overrightarrow{<T_i>}) = (\overrightarrow{T_j\ f_j}) \quad \Gamma \vdash_\exists e_2 : T \quad T \preceq T_k}{\Gamma \vdash_\exists (e_1.f_k = e_2) : T_k} \text{ (T-FIELD-ASS)}$$

Figure 2: Typing rule for field assignment (supplemental to Figure 1).

here. We assume the definition of subtyping $\preceq$ on (non-quantified) types as in FJ [1, 3], which is essentially the reflexive, transitive closure of the declared subclassing relationship in the program.

Type assignment is then defined by the derivation system of Figure 1.

Crucially, we have the following result for this type system (modulo the well-formedness considerations, which we are eliding here).

THEOREM 5 (SOUNDNESS). *For any closed expression e and type T, if $\emptyset \vdash_\exists e : T$ then either e diverges, or there exists a (unique) value v such that $e \rightarrow v$ then $\emptyset \vdash_\exists v : T$.*

The most interesting aspect of this result is that it does *not* extend to the Java language itself. Indeed, if one takes an analogous approach to a language or calculus with state, the approach to existential types presented above is naturally unsound. The reason for this can be understood by further analysis of the ($\exists\mathcal{E}$) rule above. The assumption implicit in this rule is that, while we cannot know the witness type for e_1, it is some (fixed) type, which we give a fresh name Y to. However, since this name is used to type *every* occurrence of e_1 in the resulting expression, this is unsound if the witness type can be different for different occurrences. In particular, if it is possible to modify the state on which e_1 depends between the evaluations of different occurrences, then to assume that all such occurrences have a fixed witness type Y is dangerous.

For a concrete example, we need to informally consider the extension of the calculus to include a heap, object references and field assignments of the form $(e.f = d)$. The semantics of the language must be significantly extended to deal with such features, but we can illustrate the point at a high level here. The analogous additional type assignment rule for field assignment would be that shown in Figure 2.

Consider now the following (Java) class definition, defining a list whose elements may each be of a different (but parameterised) type:

```
class VariedList <X> {
  X item;
  VariedList <?> next;
}
```

Recall that the Java type VariedList<?> corresponds with the existential type $\exists X.\mathsf{VariedList}\langle X\rangle$. In particular, it is possible to assign an expression of type VariedList < T > for any type T, to the next field of an object of this class.

Suppose now that we have a heap containing a "list" of four elements (i.e., the next field of the first object points to the second object, etc.), which are alternately of types VariedList < Integer > and VariedList < String >. Furthermore, assume that every item field of the four objects is non-null. Let x be a variable referring to the first element of the list (which is a VariedList < Integer > object).

Let e_1 be the expression $(x.\mathsf{next} = x.\mathsf{next}.\mathsf{next})$ and let Γ be the typing context $\{x : \mathsf{VariedList}\langle\mathsf{Integer}\rangle\}$. Using the typing rules above, it is possible to derive the type judgement $\Gamma \vdash_\exists e_1 : \exists Y.\mathsf{VariedList}\langle Y\rangle$ Then, crucially using the ($\exists\mathcal{E}$) rule, it it possible to derive the judgement:

$$\Gamma \vdash_\exists (e_1.\mathsf{item} = e_1.\mathsf{item}) : \exists Z.\mathsf{VariedList}\langle Z\rangle$$

In particular, the expanded expression $((x.\mathsf{next} = x.\mathsf{next}.\mathsf{next}).\mathsf{item} = (x.\mathsf{next} = x.\mathsf{next}.\mathsf{next}).\mathsf{item})$ is typeable. But this is unsound: when this expression is executed in the heap described, it will result in the assignment of a String to an Integer field.

Acknowledgements

We are extremely grateful to Nicholas Cameron, Mariangiola Dezani and especially Sophia Drossopoulou for encouragement, generous discussions and invaluable input to this paper.

This work was funded in part by the IST-2005-015905 MOBIUS project.

3. REFERENCES

[1] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM ToPLAS*, 23(3):396–450, 2001.

[2] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988.

[3] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[4] Mads Torgersen, Christian Plesner Hansen, Erik Ernst, Peter von der Ahé, Gilad Bracha, and Neal Gafter. Adding wildcards to the java programming language. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 1289–1296, New York, NY, USA, 2004. ACM.