Variant Ownership with Existential Types

Nicholas Cameron
Imperial College London
ncameron@doc.ic.ac.uk

Sophia Drossopoulou
Imperial College London
scd@doc.ic.ac.uk

Abstract

We propose an ownership types system with existential quantification of owners, similar to the existential quantification of types in models of Java wildcards. This produces a system with variant ownership types. Using explicit existential types for variance is more expressive, less ad hoc, and easier to understand and reason about than previous solutions. Furthermore, we propose using both type and ownership parameters to increase the precision with which variant types can be specified.

1. Background

Ownership types (in all their various flavours, e.g., [5]) allow for the structuring of objects in the heap according to some tree or graph. Each object has an owner, denoted as part of the object's type. A type may also be parameterised by *contexts*, which may be used as owners within the class definition. For example, consider a list class:

```
class List<o1, o2> {
  Object<o2> datum;
  List<o1,o2> next;
}
```

The formal parameter `o1` denotes the formal owner of the class; it is used as the actual owner of the `next` field. `o2` is a formal context parameter of `List`; it is used as the owner of `datum` and as an actual context parameter of `next`.

In general, subtyping is invariant with respect to ownership (more precisely, the *inside* relation over owners). That is, given two types `C<o1>` and `C<o2>`, even if `o2` is owned by (i.e., is inside) `o1`, `C<o2>` is not a subtype of `C<o1>`, nor is `C<o1>` a subtype of `C<o2>`.

Some form of ownership variance is often safe and desirable; as found in a restricted form in several systems, e.g., '`any`' in the universes type system [6], `?` in MOJO [2], and by using variance annotations [7].

Existing ownership systems are either invariant or offer some form of more or less ad hoc variance. Existential types offer a way to implement variance in a uniform and theoretically well understood manner.

Existential types were first proposed as a mechanism to model abstraction and data hiding (e.g., in module systems) [8]. An existential type is a polymorphic type where a type variable is existentially quantified. An existential type `∃X.X` can be read as 'there exists some type `X`', and the type variable can be thought of as *hiding* some concrete type.

Existential types have been used in a slightly altered form in object-oriented programming languages for subtype variance, i.e., to soften the mismatch between parametric (generics) and inclusion (subclassing) polymorphism. Similarly to owners, generic types are invariant, that is `List<Fish>` is not a subtype of `List<Animal>`, even if `Fish` is a subtype of `Animal`. Java wildcards use implicit existential types to allow variance in a type's parameters; for example, allowing `List<? extends Fish>` to be a subtype of `List<? extends Animal>`.

2. Language Design

We propose adding existential quantification of owners to ownership types to implement owner variance. So, for example, `∃o.Animal<o>` denotes an `Animal` object that is owned by *some* owner. We use bounds that specify where the owner must exist within the topology of the heap; for example `∃o → [a b].Animal<o>` restricts `o` to be owned by `b` and to own[1] `a`. We omit bounds and empty parameter lists in the following examples for clarity.

We also allow parameterisation of types, classes and methods by types; this lets us express more of the interesting relationships between owners. For example, we can define the `List` class using generics:

```
class GenericList<owner, X> {
  X datum;
  GenericList<owner, X> next;
}
```

We can then use the type `GenericList<this, Animal<o3>>` to denote a list owned by `this`, where each item in the list is an `Animal` owned by `o3`. With existential ownership types this expressivity is extended further: `∃o.GenericList <o, Animal<this>>` denotes a list owned by *some* owner where each element is owned by `this`; `∃o1,o2.GenericList <o1, Animal<o2>>` denotes a list owned by some owner where each element is owned by some owner (but each element is owned by the same owner); `∃o1.GenericList<o1, ∃o2.Animal<o2>>` denotes a list where each element is owned by some owner and the owner of each element may be different. To the best of our knowledge such expressiveness is not found elsewhere.

By using existential types we can also denote the type `∃o.List<o, o>`. This describes a list where the list itself and its data are owned by the *same* unknown owner. Such types can not be denoted using variance annotations. However,

[Copyright notice will appear here once 'preprint' option is removed.]

[1] In both cases, we mean either directly or transitively owned.

$$
\begin{array}{lll}
\mathtt{e} & ::= & \mathtt{x} \mid \mathtt{null} \mid \mathtt{x.f} \mid \mathtt{x.f\ =\ e} \mid \mathtt{x.<}\overline{\mathtt{a}}\mathtt{,\ }\overline{\mathtt{T}}\mathtt{>m(}\overline{\mathtt{e}}\mathtt{)} \mid \mathtt{new\ C<}\overline{\mathtt{a}}\mathtt{,\ }\overline{\mathtt{T}}\mathtt{>} \mid \\
 & & \mathtt{open\ e_1\ as\ x,}\overline{\mathtt{o}}\mathtt{\ in\ e_2} \mid \mathtt{close\ e\ with\ }\Delta\mathtt{\ hiding\ }\overline{\mathtt{a}} \qquad \textit{expressions} \\
\mathtt{v} & ::= & \mathtt{close\ v\ with\ }\Delta\mathtt{\ hiding\ }\overline{\mathtt{r}} \mid \iota \mid \mathtt{null} \qquad \textit{values} \\
\mathtt{Q} & ::= & \mathtt{class\ C<}\Delta\mathtt{,\ }\overline{\mathtt{X}}\mathtt{>}\ \{\overline{\mathtt{T\,f;}}\ \overline{\mathtt{W}}\} \qquad \textit{class declarations} \\
\mathtt{W} & ::= & \mathtt{<}\Delta\mathtt{,\ }\overline{\mathtt{X}}\mathtt{>\ T\,m(}\overline{\mathtt{T\,x}}\mathtt{)}\ \{\mathtt{return\ e;}\} \qquad \textit{method declarations} \\
 & & \\
\mathtt{N} & ::= & \mathtt{C<}\overline{\mathtt{a}}\mathtt{,\ }\overline{\mathtt{T}}\mathtt{>} \qquad \textit{class types} \qquad \mathtt{a} ::= \bigcirc \mid \mathtt{o} \mid \mathtt{owner} \mid \mathtt{x} \quad \textit{actual owners} \\
\mathtt{M} & ::= & \mathtt{N} \mid \mathtt{X} \qquad \textit{non-existential types} \qquad \mathtt{b} ::= \mathtt{a} \mid \bot \quad \textit{bounds} \\
\mathtt{T} & ::= & \mathtt{M} \mid \exists\Delta\mathtt{.N} \qquad \textit{types} \qquad \Delta ::= \overline{\mathtt{o}\!\to\![\mathtt{b}_l\ \mathtt{b}_u]} \quad \textit{owner environments}
\end{array}
$$

Figure 1. Syntax of Jo∃ expressions and types.

if a language includes owner polymorphic methods (used to support programming idioms such as iterators [4]), then such types can occur during execution of programs. Therefore, existential owners can support owner polymorphic methods, whereas variance annotations on owners [7] can not. A similar problem motivates the use of existential types to model Java wildcards [1].

3. Formalisation

To formalise the notion of existential types for owner variance, we extend a 'traditional' single ownership calculus (e.g., [5]); the result is our minimal language, Jo∃. We add existential quantification of owners to the syntax of types, and explicit `open` and `close` expressions. In this way, we follow the more traditional model of existential types [3], rather than the 'wildcards' approach of using implicit packing and unpacking of existential types, present in subtyping and capture conversion [1]. In Jo∃ there is a strong distinction between types and parameters that may be existentially quantified; thus, the loss of expressivity found when modelling Java wildcards with explicit packing and unpacking [3] is avoided.

The syntax of Jo∃ is given in Fig. 1; we elide most runtime syntax and some other detail. Most of the expression syntax is as might be expected for a Java-like language. We add `open` and `close` expressions to eliminate and introduce existential types. If $\mathtt{e}_1$ in the `open` expression has existential type, e.g., `∃o.C<o>`, then, within the scope of $\mathtt{e}_2$, the programmer may use `o` as a formal owner and `x` as a variable with type `C<o>`. Thus, an expression with existential type may be used as a non-existentially typed expression; this is similar to *capture conversion* of wildcard types in Java [3].

The `close` expression wraps a sub-expression (`e`) with an existential type by hiding some of the owners present in `e`'s type. For example, if `e` has type `C<this>`, then the expression `close e with o hiding this` has the existential type `∃o.C<o>`.

Note that the receiver of field access or method invocation must be a variable (`x`, which includes `this`). Thus we can easily substitute `x` for `this` when type checking these expressions. Expressivity is not lost since the programmer can always use the `open` expression with empty $\overline{\mathtt{o}}$ to act as a 'let' expression.

Actual owners (`a`) may be the distinguished 'world' owner (◯), formal owners (`o`), the object's owner (`owner`), or variables (`x`). Bounds on formal owners also allow the 'bottom' owner, that is the owner that is owned by all objects; it is used to indicate an owner variable without a lower bound.

Subtyping in Jo∃ follows the full variant of System $F_{<:}$ with existential types and ∃J [3]. Existential types are subtypes where the owners in the subtype are more strict than the owners of the supertype. For example, `∃o → [⊥ this].C<o>` is a subtype of `∃o → [⊥ ◯].C<o>`, since `this` is inside ◯. Non-existential types remain invariant. Such subtyping gives the variance properties that motivate this work.

4. Conclusion and Future Work

We have shown how existential types may be used for variance of ownership types. Compared to other systems, we aim to make Jo∃ less ad hoc and more expressive, and to make the existential type apparatus more explicit. This explicit use of existential types and `open` and `close` expressions will make reasoning about properties of the language simpler, and the connection to previous work more explicit. By using existential types we also allow owner polymorphic methods to be handled correctly.

We have a complete formal definition of Jo∃ and, based on our earlier work [3], expect this to be sound. We aim to prove the owners-as-dominators (AKA, deep ownership) property for Jo∃. Completion of the proofs of soundness and ownership properties are the next step for us; extensions to the work could be extending Jo∃ to a multiple ownership system [2], or to other ownership systems such as ownership domains or universes [6].

References

[1] Nicholas Cameron, Sophia Drossopoulou, and Erik Ernst. A Model for Java with Wildcards. In *ECOOP*, 2008.

[2] Nicholas Cameron, Sophia Drossopoulou, James Noble, and Matthew Smith. Multiple Ownership. In *OOPSLA*, 2007.

[3] Nicholas Cameron, Erik Ernst, and Sophia Drossopoulou. Towards an Existential Types Model for Java Wildcards. In *FTfJP*, 2007.

[4] Dave Clarke. *Object Ownership and Containment.* PhD thesis, UNSW, Australia, 2002.

[5] David G. Clarke, John M. Potter, and James Noble. Ownership Types for Flexible Alias Protection. In *OOPSLA*, 1998.

[6] W. Dietl, S. Drossopoulou, and P. Müller. Generic Universe Types. In *ECOOP*, 2007.

[7] Yi Lu and John Potter. On Ownership and Accessibility. In *ECOOP*, 2006.

[8] John C. Mitchell and Gordon D. Plotkin. Abstract Types have Existential Types. In *POPL*, 1985.