Model abstraction and data hiding (e.g., in module systems) is theoretically well understood manner. Some form of more or less ad hoc variance. Existential types offer a way to implement variance in a uniform 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., 4) 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 the list class:

```java
class List<o> {
  o: Object<> datum;
  owner: List<o> next;
}
```

The keyword owner denotes the formal owner of the class; it is used as the actual owner of the next field. o 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, ownership (more precisely, the inside relation over owners) is invariant with respect to subtyping. That is, given two types o1:C and o2:C, even if o2 is owned by (i.e., is inside) o1, o2:C is not a subtype of o1:C, nor is o1:o a subtype of o2:C.

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 [5], ? in MOJO [2], and by using variance annotations [6].

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) [7]. An existential type is a polymorphic type where a type variable is existentially quantified. An existential type $\exists X.X$ 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, $\exists o.o:Animal$ 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 $\exists o \rightarrow [a b].o:Animal$ restricts o to be owned by b and to own a. We omit bounds and empty parameter lists in the 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:

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

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

1 It is important to note that this type means $\exists o.(o:Animal)$ and not $\exists (o.o):Animal$: this is important when considering how the owner is propagated into types used in the class definition.

2 In both cases, we mean either directly or transitively owned.
We do not support existential quantification of types. A system supporting existential quantification of types as well as of owners, would allow very powerful types, e.g.,

\[
\exists o_1, o_2, X \rightarrow [\bot, o_2: \text{Animal}, o_1: \text{GenericList}\langle X \rangle]
\]

We are not sure whether such types are of practical use and believe most issues are orthogonal to this work; we relegate their study to further work.

3. Formalisation

To formalise the notion of existential types for owner variance, we extend a ‘traditional’ single ownership calculus (e.g., [4]); the result is our minimal language, Jo3. 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 Jo3 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 Jo3 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 \(e\) in the open expression has existential type, e.g., \(\exists o. o:C\), then, within the scope of \(e\), the programmer may use \(o\) as a formal owner and \(x\) as a variable with type \(o:C\). 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 this:C, then the expression close e with o hiding this has the existential type \(\exists o. o:C\).

As might be expected, the operational semantics of Jo3 includes a reduction rule to reduce together an open and close expression. Thus, open (close e with o hiding this) as x in e reduces to [this/o, e/this[e]].

Note that the receiver of field access or method invocation must be a variable (x, which includes this). Thus we can substitute x for this when type checking these expressions, and do not require some form of path types. Expressivity is not lost since the programmer can always use the open expression with empty \(\Sigma\) to act as a ‘let’ expression.

Actual owners (a) may be the distinguished ‘world’ owner (\(\circ\)), 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 Jo3 follows the full variant of System \(F_\omega\) with existential types and \(\exists J\) [3]. Existential types are subtypes when the owners in the subtype are more strict than the owners of the supertype. For example, \(\exists o \rightarrow [\bot, \text{this}]. o:C\) is a subtype of \(\exists o \rightarrow [\bot]. o:C\), since this is inside \(\circ\). 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 Jo3 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. Systems that include variance annotations can express types using polymorphic methods that can not be denoted without explicit existential types.

We have a complete formal definition of Jo3 and, based on our earlier work [3], expect this to be sound. We aim to prove the owners-as-dominated (AKA, deep ownership) property for Jo3. Completion of the proofs of soundness and ownership properties are the next step for us; extensions to the work could be extending Jo3 to a multiple ownership system [2], or to other ownership systems such as ownership domains or universes [5].

References