Gradual Ownership Types

Ilya Sergey and Dave Clarke

IBBT-DistriNet, Department of Computer Science,
Katholieke Universiteit Leuven, Belgium
{firstname.lastname}@cs.kuleuven.be

Abstract. Gradual Ownership Types are a framework allowing programs to be partially annotated with ownership types, while providing the same encapsulation guarantees. The formalism provides a static guarantee of the desired encapsulation property for fully annotated programs, and dynamic guarantees for partially annotated programs via dynamic checks inserted by the compiler. This enables a smooth migration from ownership-unaware to ownership-typed code.
The paper provides a formal account of gradual ownership types. The theoretical novelty of this work is in adapting the notion of gradual type system with respect to *program heap properties*, which, unlike types in functional languages or object calculi, impose restrictions not only on data, but also on the environment the data is being processed in. From the practical side, we evaluate applicability of Gradual Ownership Types for Java 1.4 in the context of the Java Collection Framework and measure the necessary amount of annotations for ensuring the owners-as-dominators invariant.

1 Introduction

Type systems for ownership in object-oriented languages provide a declarative way to statically enforce a notion of object encapsulation in object-oriented programs. Object ownership ensures that objects cannot escape from the *scope* of the object or collection of objects that *own* them. Variants of ownership types allow a program to enjoy such computational properties as data race-freedom [4], disjointness of effects [8], various confinement properties [28] and effective memory management [5].

However, there are several obstacles to the adoption of ownership types. The first is verbosity. One way to overcome this problem is to omit annotations and use type inference instead. Unlike traditional type systems, ownership annotations are mostly design-driven, thus full inference of ownership types is not particularly useful, since a correct, trivial ownership typing always exists [12]. Therefore, inference is only practically applicable when some annotations are already provided to indicate the programmer's intention. But even in the case of partially-annotated programs, ownership type inference tends to produce an excessive amount of inferred annotations [21] or imprecise results due to the conservatism of the underlying analysis [20]. The second obstacle is that ownership types are often too rigid to capture the dynamic evolution of an object graph in real applications, and in some cases the imposed constraints need to be relaxed.

Adding ownership annotations into the code is similar to the migration from the untyped to typed code, a topic of much research nowadays [15, 17, 26, 27]. Complete absence of types facilitates the fast prototyping and rapid evolution of the system, so one might need to introduce types into the code only when the demands for reliability and performance of the program are established. Ownership types provide more fine-grained safety guarantees. In this respect refactoring a program to employ them can be considered as a migration from typed to "even more typed" code. This observation leads to the idea of applying a *gradual* approach for an incremental migration.

This work is based on the ownership type system of Clarke and Drossopoulou [8], which ensures the *owners-as-dominators* invariant. It is expressive enough to investigate the concepts of interest and is close enough to a real language to guide the implementation. Nevertheless our approach is idiomatic and can be applied to many other ownership type systems. Overall, this paper makes the following contributions:

- A type system for a Java-like object-oriented language providing gradual ownership types, enabling the migration from ownership-unaware to ownership-annotated code.
- A type-directed program translation that ensures the dynamic preservation of the ownership invariant when insufficient type annotations are provided; soundness theorems and properties of the type-directed translation.
- An implementation of a translating compiler for full Java 1.4 that supports gradual ownership types and provides hints for smooth program migration.
- A report on migrating classes from Java's SDK to use gradual ownership types.
- A discussion on extending the described framework for different existing ownership policies and an overview possible design choices of the implementation.

2 Intuition and overview

This section gives the essence of ownership types enforcing the *owners-as-dominators* policy (OAD) and provides some intuition on the "gradualization" of the type system.

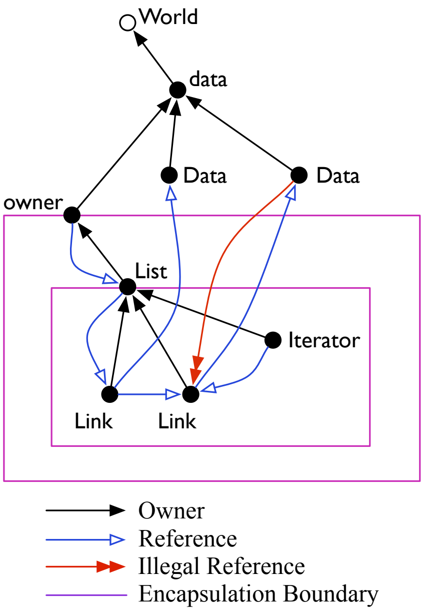
Ownership types are based on a *nesting* relation on objects ($\prec$). At run-time, each object o has an *owner*, i.e., another object o', such that $o \prec o'$. Nesting is a tree-shaped partial order on objects with greatest element **`world`**. The OAD invariant is as follows: *Given an object o and its* owner *o', every path in the object graph of a program from the* program roots *along object fields that ends in o, contains o'*. I.e., there are no fields referring to o that bypass o'. This means that one object cannot refer to a second object directly as a field, unless the first object is *inside* the second object's owner.

Figure 1 gives an example of the class `List` using ownership types. The class carries two *ownership parameters*: `owner` and `data`. The first parameter, `owner`, refers to the `List` instance's immediate, or *primary*, owner. The second parameter, `data`, refers, by conventions of the type system, to some object *outside* or equal to `owner`. As usual, **`this`** refers to the current instance itself. The same reasoning is applicable to two auxiliary classes, `Link` and `Iterator`. In the `List`'s method `add()`, the programmer indicates, by creating an instance of the class `Link` with owners **`this`** and `data` respectively, that this particular instance of `Link` is nested within its creator instance `List` and the content of the link can be accessed only through the owner referred to as `data` in `List`. The same is true for the instance of the class `Iterator`.

Ownership information for our list example can be provided by only five annotations. Three class parametrizations name the owners of the class instances and two

Fig. 1 A motivating example and the design intention: a list and its iterator code with structural (underlined) and constraint (grayed) ownership annotations.

```
class List<owner, data> {
  Link <this, data> head;
  void add(Data <data> d) {
    head = new Link<this, data>(head, d);
  }
  Iterator <this,data> makeIterator() {
    return new Iterator<this, data>(head);
  }
}
class Link<owner,data> {
  Link <owner, data> next;
  Data <data> data;
  Link(Link <owner, data> next, Data <data> data) {
    this.next = next; this.data = data;
  }
}
class Iterator<owner, data> {
  Link <owner, data> current;
  Iterator(Link <owner, data> first) {
    current = first;
  }
  void next() { current = current.next; }
  Data <data> elem() { return current.data; }
  boolean done() { return (current == null); }
}
```



allocation sites provide concrete owners for created objects. These annotations, underlined in Figure 1, are *structural*: they declare the information about nesting of objects involved (i.e., **this** ≺ owner ≺ data ≺ **world**) and define the owners of new instances. The remaining, *constraint*, annotations, grayed in the code, "propagate" ownership information through the program, since mutable variables and fields are traditionally annotated with types to keep information about objects they point to. We require the first kind of annotations to be explicitly specified, because it (a) reflects the programmer's intentions with respect to the invariant and (b) enables a simpler implementation of run-time dynamic checking—no ownership information needs to be inferred dynamically.

The runtime checking of conformance of an object's ownership structure to the expectation is performed via dynamic type casts. This technique is typical for gradual approaches: when an untyped value is coerced to a typed value, a dynamic check is performed to ensure that the further interactions through this particular reference conform to the target's type contract, in this setting, its ownership type. However, the preservation of the OAD invariant requires not only conformance of *actual* and *expected* types, but also checking that the *nesting* constraints are preserved—this information is lost when ownership information is lost.

The only place where the owners-as-dominators invariant can actually be broken is by a bad field assignment, which makes field assignments good candidates for extra run-time checks. Consider the following assignment:

```
receiver.f = result;
```

The correctness requirement for such an assignment demands that `receiver` $\prec$ `o`, where `o` is an owner of the object referred to by `result`. If the declaration of the field `f` lacks ownership information, there is a chance that the OAD invariant will be violated since the type of `f` may no longer impose any nesting between `receiver` and `result`. This is a sort of contract that should be checked dynamically. We call these *boundary checks*.

One can notice that dynamic type casts operate with objets' ownership structures, whereas boundary checks traverse a part of the heap and, therefore, are significantly more expensive. However, performing type casts *before* boundary checks might help to avoid most of them, since after the check we gain some extra knowledge of an object's structure. This observation leads us to a two-staged, typed-directed transformation, where each stage uses the available type information to perform one sort of check: *type conformance* and *nesting*. In the following sections we develop a staged algorithm for the correct translation. The first pass will insert dynamic casts and the second will handle possible OAD violations by inserting boundary checks.

Defined, unknown and dependent owners An important part of the ownership type system is the static representation of owners. The example in Figure 1 demonstrated one usage of ownership class parameters. The following example exhibits the concept of *dynamic aliasing* [8], which employs local final variables as local owners:

```
final List<p, world> list = new List<p, world>();
Iterator<list, world> iter = list.makeIterator();
```

Variable `list` denotes the owner of the iterator in `iter`. When `list` goes out of scope, the type `Iterator<list, world>` and other types containing owner `list` become illegal.

Following gradual types we introduce a notion of the special *unknown owner* "?". Types annotated with "?" in a gradually-typed language defer the checking of types to run-time via checks inserted by the compiler. In our system, types with no annotations are just syntactic sugar for types with all ownership annotations unknown, e.g., `List` $\equiv$ `List<?,?>`. The following code gives the essence of unknown owners:

```
List list; // ≡ List<?,?>
list = new List<p, world>();
list = new List<this, world>();
List<p, world> newList = list; // inserted cast (List<p, world>)list
```

The first two assignments are valid since the type of `list` does not specify which objects must own the instance referred by the variable. The last assignment is valid too; however, it requires a dynamic cast, due to the type refinement `List<?, ?>` $\Rightarrow$ `List<p, world>` to make sure that the owners of `list` matches the specification of `newList`.

Information lost due to unknown owners can be partially regained by tracking of dependencies between immutable references and owners of objects they refer to. For this purpose we introduce *dependent owners*, which record the origin of some owner arguments, allowing one to check them for equality without knowledge about the nesting. Figure 2 provides and example with dependent owners. Class `E` declares a field of type `D`. However, information about the owner of the object referred to by field `myD` is lost due to the missing ownership annotation in the field declaration on line 1. As a consequence, the owner of variable `d` in line 9 is unknown. Nevertheless, since `d` is final, one can see that the owner of the object referred by `d` is the same as the one expected as

Fig. 2 Dependent owners in action

```
1  class E<P> { D myD = ... }
2
3  class D<owner> {
4    E<owner> e;
5    void use(D<owner> arg) { ... }
6    void exploit(E<owner> arg) { this.e = arg; }
7    void test(E e) {
8      final D d = e.myD; // implicitly, d: D<d^{D.owner}>
9      d.use(d); // type refinement, but no type cast required
10     d.exploit(e); // type refinement, dynamic type cast required
11   }
12 }
```

of a parameter of the instance method `d.use()`. This knowledge is preserved by assigning the type `D<d`$^{D.owner}$`>` to variable `d`. This should be read as "`d` has the type `D` and the owner of the object referred to is locally denoted as `d`$^{D.owner}$". The superscript `D.owner` refers to the particular ownership parameter of the statically known type `D`. Thus, by equality of owners, no extra dynamic check is required in line 9. Still, the owner of `e` remains unknown, so the method call `d.exploit()` on line 10 is potentially dangerous due to type refinement, and therefore the cast `E<?>` $\Rightarrow$ `E<d`$^{D.owner}$`>` is required.[1]

3 The language JO$_?$

To investigate the meta-theory of gradual ownership types we define JO$_?$, an imperative Java-like language, extended with ownership types, and unknown and dependent owners, based on the system JOE$_1$ by Clarke and Drossopoulou [8].

3.1 Syntax

Figure 3 provides the full syntax of JO$_?$. A program in JO$_?$ is a collection of classes. A class definition describes a class named c, parametrized by the ownership parameters $\alpha_{i\in 1..n}$ with the superclass c', whose ownership parameters are instantiated with $p_{j\in 1..n'}$.[2] Methods have only one parameter for simplicity. Expressions in JO$^+_?$ are a-normal form (ANF), i.e., all intermediate computations are named and assigned to the immutable variables. Local variables can be used as owners, as long as they do not escape the scope of a local stack frame.

Types and owners A type $c\langle p_{i\in 1..n}\rangle$ consists of a class name c and a vector of ownership arguments $p_{i\in 1..n}$. Owners are represented syntactically by owner and term variables (α and x, respectively), *dependent* owners and *run-time* owners such as **`world`** and heap locations (i.e, run-time object identifiers). $x^{c.i}$ denotes the dependent owner corresponding to the i-th ownership parameter of the object referred to by the term

[1] We have chosen the term "dependent owners" because of similarity of the idea to the notion of *path-dependent types* [22]—the value of the owner depends on the value of an object.

[2] More expressive possibilities exist in the literature, for example, by allowing the programmer to declare the expected relationship between owner parameters to a class [10].

Fig. 3 Syntax of $\mathsf{JO}_?$ and syntactic helper functions

P	$::= \mathit{class}_{j\in 1..m}$	***programs***
class	$::= \texttt{class}\ c\langle\alpha_{i\in 1..n}\rangle\ \texttt{extends}\ c'\langle p_{j\in 1..n'}\rangle\ \{\mathit{fd}_{k\in 1..m};\ \ \mathit{meth}_{l\in 1..u}\}$	***class declarations***
fd	$::= t\ f$	***field declarations***
meth	$::= t\ m(t\ x)\ \{e\}$	***method declarations***
e	$::= x \mid \texttt{let}\ x = b\ \texttt{in}\ e$	***expressions***
b	$::= x.f \mid x.f = x \mid x.m(x) \mid \texttt{new}\ c\langle p_{i\in 1..n}\rangle \mid \texttt{null}$	***computations***
v	$::= \iota \mid \texttt{null}$	***values***
E	$::= \emptyset \mid E, x : t \mid E, \iota : t \mid E, p \prec p'$	***typing environments***
B	$::= \emptyset \mid B, \alpha = k \mid B, x = v$	***bindings***

k	$::= \texttt{world} \mid \iota$	***run-time owners***	$x, y, z, \texttt{this}$	***variables***
p, q	$::= x \mid \texttt{this} \mid k \mid ? \mid \alpha$	***owners***	ι	***heap locations***
t, s	$::= c\langle p_{i\in 1..n}\rangle$	***types***	α	***formal owners***
o	$::= \langle c\langle k_{i\in 1..n}\rangle, \overline{f \mapsto v}\rangle$	***objects***	$x^{c.i}$	***dependent owners***
H	$::= \overline{\iota \mapsto o}$	***heaps***	$?$	***unknown owners***

$$
\begin{array}{rlrl}
\mathsf{defined}(p) \triangleq (p \neq ?) \wedge (p \neq x^{c.i}) & & \mathsf{owner}(c\langle\rangle) \triangleq \texttt{world} & \\
\mathsf{undefined}(p) \triangleq \neg\mathsf{defined}(p) & & \mathsf{owner}(c\langle p_{i\in 1..n}\rangle) \triangleq p_1, \text{ where } n > 0 & \\
\mathsf{actual}(p) \triangleq (p = \texttt{world}) \wedge (p = \iota) & & \mathsf{owner}_j(c\langle p_{i\in 1..n}\rangle) \triangleq p_j, \text{ where } 0 < j \leq n & \\
\mathsf{arity}(c) \triangleq n, \text{ s.t. } \texttt{class}\ c\langle\alpha_{i\in 1..n}\rangle\ \in\ P & & \mathsf{owners}(c\langle p_{i\in 1..n}\rangle) \triangleq p_1 \ldots p_n &
\end{array}
$$

variable x, whose statically known class type is c. Dependent owners are not supposed to be specified by the programmer. Instead, they are inferred by the compiler. We often use an alternative notation $c\langle\sigma\rangle$ for a type $c\langle p_{i\in 1..n}\rangle$, assuming σ to be a substitution $\{\alpha_i \mapsto p_i \mid i \in 1..\mathsf{arity}(c)\}$, and α_i are formal ownership parameters of the class c.

To distinguish between different kinds of owners when checking the well-formedness of types, we introduce several syntactic helper functions (Figure 3).

Objects and heaps In addition to having the class name and field values, an object also has a binding for its owner parameters, either `world` or some non-null heap locations. A heap H is a partially defined map from locations to objects.

3.2 Environments and owners

A typing environment E binds variables and heap locations with types and defines ordering assumptions on owners with respect to the nesting relation $\prec$. The bindings B map formal owners to run-time owners and variables to values.

The dynamic semantics is defined in Section 5 in terms of an explicit binding of free variables, rather than via substitution. The presence of binding environment in the typing judgements of the form $E; B \vdash \mathfrak{F}$ for some succedent $\mathfrak{F}$ does not affect the static semantics of $\mathsf{JO}_?$, but we will need it to establish equalities between typing environments and dynamic bindings in the proof of the type preservation theorem.

A typing environment E is *well-formed* if $\prec$ is *antisymmetric* on $\{p \mid p \in \mathrm{dom}(E)\}$, i.e, the environment does not introduce cycles in ownership. Well-formed *environment-*

Fig. 4 Well-formed owners and owner nesting

$$\boxed{E;B \vdash p}$$

$$\text{(OWN-WORLD)}\ \frac{E;B \vdash \diamond}{E;B \vdash \texttt{world}} \qquad \text{(OWN-VAR)}\ \frac{\begin{array}{c}E;B \vdash \diamond \\ E;B \vdash x : t\end{array}}{E;B \vdash x} \qquad \text{(OWN-VAL)}\ \frac{\begin{array}{c}E;B \vdash \diamond \\ E;B \vdash \iota : t\end{array}}{E;B \vdash \iota} \qquad \text{(OWN-?)}\ \frac{E;B \vdash \diamond}{E;B \vdash ?}$$

$$\text{(OWN-DEPENDENT)}\ \frac{\begin{array}{c}E;B \vdash x \\ i \in 1..\mathsf{arity}(c)\end{array}}{E;B \vdash x^{c.i}} \qquad \text{(OWN-IN)}\ \frac{\begin{array}{c}E;B \vdash \diamond \\ p \prec p' \in E\end{array}}{E;B \vdash p, p'}$$

$$\boxed{E;B \vdash p \prec p'}\ \mathsf{defined}(p), \mathsf{defined}(p')$$

$$\text{(IN-ENV)}\ \frac{p \prec p' \in E}{E;B \vdash p \prec p'} \qquad \text{(IN-REFL)}\ \frac{E;B \vdash p}{E;B \vdash p \prec p} \qquad \text{(IN-TRANS)}\ \frac{\begin{array}{c}E;B \vdash p \prec p' \\ E;B \vdash p' \prec p''\end{array}}{E;B \vdash p \prec p''} \qquad \text{(IN-VAR)}\ \frac{\begin{array}{c}E;B \vdash x : t \\ p = \mathsf{owner}(t)\end{array}}{E;B \vdash x \prec p}$$

$$\boxed{E;B \vdash p \lesssim p'}$$

$$\text{(SUB-LEFT)}\ \frac{\begin{array}{c}E;B \vdash p \quad E;B \vdash q \\ \mathsf{undefined}(q)\end{array}}{E;B \vdash p \lesssim q} \qquad \text{(SUB-RIGHT)}\ \frac{\begin{array}{c}E;B \vdash p \quad E;B \vdash q \\ \mathsf{undefined}(q)\end{array}}{E;B \vdash q \lesssim p} \qquad \text{(SUB-INCL)}\ \frac{E;B \vdash p \prec p'}{E;B \vdash p \lesssim p'} \qquad \text{(SUB-WORLD)}\ \frac{E;B \vdash p}{E;B \vdash p \lesssim \texttt{world}}$$

binding pairs ($E;B \vdash \diamond$) are omitted and can be found in the companion technical report [25]. Informally, the pair $E;B$ enables owners and types in E to be used modulo equalities in the run-time binding environment B. To keep the presentation tractable, we omit explicit mentioning of the rules dealing with such equalities. The *well-formed owner* relation ($E;B \vdash p$) is shown in Figure 4. The rules (OWN-DEPENDENT) and (OWN-?) are novel for the gradual type system. A dependent owner is well-formed if the corresponding variable is in scope and if i does not exceed the ownership-arity of the class c. The definition of the nesting relation on owners (Figure 4, $E;B \vdash p \prec p'$) captures only defined owners. It is then embedded into a more general *consistent-inside* relation ($E;B \vdash p \lesssim p'$), which deals also with dependent and unknown owners. Informally, no precise information about nesting can be retrieved from unknown or dependent owners. Note that $\lesssim$ is not transitive, so $E;B \vdash q \lesssim ?$ and $E;B \vdash ? \lesssim p$ do not imply $E;B \vdash q \lesssim p$ for any *defined* p and q.

To state the OAD invariant we need a definition of a heap flattening. The notation $\widehat{H}$ is used also to *flatten* a heap H into a typing environment $\widehat{H}$.

Definition 1 (Heap flattening).

$$\widehat{H} \triangleq \{(\iota \prec o), (\iota : c\langle o, k_{i \in 2..n}\rangle) \mid (\iota \mapsto \langle c\langle o, k_{i \in 2..n}\rangle, \ldots\rangle) \in H\}$$

Definition 2 (Owners-as-Dominators Invariant [10]). $\mathsf{OAD}(H) \triangleq$ *for all locations* ι, ι' *and run-time owners* k,

$$\left.\begin{array}{c} H(\iota) = \left\langle c\langle k_{i \in 1..n}\rangle, \overline{f \mapsto v}\right\rangle \\ f_i \mapsto \iota' \ \textit{and}\ H(\iota') = \langle t', \ldots\rangle \\ \mathsf{owner}(t') = k \end{array}\right\} \Rightarrow \widehat{H}; \emptyset \vdash \iota \prec k$$

Fig. 5 Owner and type consistency; gradual subtyping

$$\boxed{E;B \vdash p \sim p'}$$

$$\textsc{(con-refl)}\ \frac{E;B \vdash p}{E;B \vdash p \sim p} \qquad \textsc{(con-right)}\ \frac{\begin{array}{c} E;B \vdash p \ \ E;B \vdash q \\ \mathsf{undefined}(q) \end{array}}{E;B \vdash q \sim p} \qquad \textsc{(con-left)}\ \frac{\begin{array}{c} E;B \vdash p \ \ E;B \vdash q \\ \mathsf{undefined}(q) \end{array}}{E;B \vdash p \sim q}$$

$$\boxed{E;B \vdash t \sim t'}$$

$$\textsc{(con-type)}\ \frac{\begin{array}{c} E;B \vdash c\langle p_{i\in 1..n}\rangle \\ E;B \vdash c\langle q_{i\in 1..n}\rangle \\ p_i \sim q_i \ \forall\, i \in 1..n \end{array}}{E;B \vdash c\langle p_{i\in 1..n}\rangle \sim c\langle q_{i\in 1..n}\rangle}$$

$$\boxed{E;B \vdash t}$$

$$\textsc{(g-type)}\ \frac{\begin{array}{c} \mathsf{arity}(c) = n \\ E;B \vdash p_1 \lesssim p_i \quad \forall i \in 1..n \end{array}}{E;B \vdash c\langle p_{i\in 1..n}\rangle}$$

$$\boxed{E;B \vdash t \lesssim t'}$$

$$\textsc{(grad-sub)}\ \frac{\begin{array}{c} E;B \vdash c\langle\sigma\rangle \leq c'\langle\sigma'\rangle \\ E;B \vdash c'\langle\sigma'\rangle \sim c'\langle\sigma''\rangle \end{array}}{E;B \vdash c\langle\sigma\rangle \lesssim c'\langle\sigma''\rangle}$$

In words, if object ι references object ι' via a field, ι must be inside the owner of ι'.

3.3 Type consistency and subtyping

Types can be constructed from any class using any owner in scope (including an unknown owner "?"), as long as the correct number of arguments are supplied and the owner (the first parameter), if present, is provably consistently-inside all other parameters. The corresponding relation $E;B \vdash t$ is defined in Figure 5.

The type consistency relation answers the question: *which pairs of static types could possibly correspond to comparable run-time types?* It allows the type checker to compare types with dependent and unknown owners. We define the type consistency relation $\sim$ on types parametrized with partially known and dependent owners via the rules in Figure 5 (the relation $E;B \vdash t \sim t'$). The definition of the subtyping $E;B \vdash t \leq t'$ is is standard for parametrized object-oriented type systems, ownership parameters are invariant [8]. In order to eliminate non-determinacy from the type-checking algorithms we need to construct a relation that combines two kinds of subsumption of types: type consistency and subtyping. This relation is used then in type rules whenever an implicit upcast is necessary [23]. Siek and Taha suggest a way to design such *consistent-subtyping* relation ($\lesssim$) for the calculus $\mathbf{Ob}_{<:}$ of Abadi and Cardelli [1]. If two types $t = c\langle\sigma\rangle$ and $t' = c'\langle\sigma''\rangle$ are related via the consistent-subtyping relation, i.e., $t \lesssim t'$, they can differ along both directions: the type consistency relation $\sim$ and the subtyping relation $\leq$. This is illustrated by the diagram on the left:

$$\begin{tikzcd} & c'\langle\sigma''\rangle \\ c\langle\sigma\rangle \arrow[ur, "\lesssim"] & \end{tikzcd} \qquad \begin{tikzcd} c'\langle\sigma'\rangle \arrow[r, "\sim"] & c'\langle\sigma''\rangle \\ c\langle\sigma\rangle \arrow[u, "\leq"] \arrow[ur, "\lesssim"'] & \end{tikzcd}$$

The "upper-left mediator" (the right part of the diagram) is a connecting link between two types. This intuition is formalized via the rule (GRAD-SUB) in Figure 5.

Fig. 6 Selected typing rules of $\mathsf{JO}_?$. Grayed parts mark explicit consistent-subtyping checks that may lead to the insertion of dynamic checks.

$$\boxed{E;B \vdash b : t}$$

$$\text{(T-NEW)}\ \frac{\begin{array}{c} E;B \vdash c\langle p_{i\in 1..n}\rangle \\ \mathsf{defined}(p_i)\ \ \forall i \in 1..n \end{array}}{E;B \vdash \mathtt{new}\ c\langle p_{i\in 1..n}\rangle : c\langle p_{i\in 1..n}\rangle} \qquad \text{(T-LKP)}\ \frac{\begin{array}{c} E;B \vdash z : c\langle\sigma\rangle \\ \mathcal{F}_c(f) = t \end{array}}{E;B \vdash z.f : \sigma_z(t)} \qquad \text{(T-LET)}\ \frac{\begin{array}{c} E;B \vdash b : t \\ E,x : \mathsf{fill}(x,t);B \vdash e : s \end{array}}{E;B \vdash \mathtt{let}\ x = b\ \mathtt{in}\ e : s}$$

$$\boxed{E \vdash t'\ m(t\ y)\{e\}}$$

$$\text{(T-UPD)}\ \frac{\begin{array}{c} E;B \vdash z : c\langle\sigma\rangle \quad \mathcal{F}_c(f) = t \\ E;B \vdash y : s \\ E;B \vdash s \lesssim \sigma_z(t) \end{array}}{E;B \vdash z.f = y : \sigma_z(t)} \qquad \text{(T-CALL)}\ \frac{\begin{array}{c} E;B \vdash y : s \quad \mathcal{MT}_c(m) = (y',t \to t') \\ E;B \vdash z : c\langle\sigma\rangle \quad E;B \vdash s \lesssim \sigma_z(t) \\ \sigma' \equiv \sigma \uplus \{y' \mapsto y\} \end{array}}{E;B \vdash z.m(y) : \sigma'_z(t')} \qquad \text{(METHOD)}\ \frac{\begin{array}{c} E,y : \mathsf{fill}(y,t) \vdash e : s \\ E \vdash s \lesssim t' \end{array}}{E \vdash t'\ m(t\ y)\{e\}}$$

3.4 Expression, method and class typing

Typing rules for expressions are described in Figure 6, following the standard approach [23]. Type rules for variables and values are standard. $m \uplus m'$ denotes the disjoint union of finite maps m and m', requiring that their domains are disjoint. σ_z is the substitution $\sigma \uplus \{\mathtt{this} \mapsto z\}$ for any substitution σ. We use the mappings $\mathcal{F}_c$ and $\mathcal{MT}_c$ for retrieving types of fields and methods of a class c. In the rules (T-LET) and (METHOD), the helper function fill converts declared types with *unknown* owners to types with *dependent* owners to track owner dependencies.

$$\mathsf{fill}(x, c\langle p_{i\in 1..n}\rangle) \triangleq c\langle q_{i\in 1..n}\rangle, \text{ where } q_i = \begin{cases} x^{c.i} & \text{if } p_i = \ ? \\ p_i & \text{otherwise.} \end{cases}$$

The definition of well-formed classes ($\vdash c$) and programs ($\vdash P;e$) is standard.

4 Type-directed translation: the language $\mathsf{JO}_?^+$

This section describes the type-based translation of programs in $\mathsf{JO}_?$ to an extended language, $\mathsf{JO}_?^+$, with run-time checks.

4.1 Syntax of $\mathsf{JO}_?^+$

The syntax is extended for dynamic type casts and boundary checks.

$$b \in \textbf{Comp} ::= \ \ldots \ \mid \ \langle t\rangle x \ \mid \ x.f \leftarrow y$$

The statement $\langle t\rangle x$ ensures that the run-time type of an object referred to by x *matches* the type t. The statement $x.f \leftarrow y$ performs the check that a field reference from x to y via the field f does not violate the ownership invariant and then performs the field update atomically. Casts and checks are not supposed to be inserted by the programmer. They are inserted instead by the compiler as described in Section 4.3.

Fig. 7 Selected typing rules of $\vdash^{C}$ and $\vdash^{C}_{\mathcal{B}}$

$$\boxed{E;B \vdash^{C} b : s}$$

$$\text{(T-CAST)}\quad \frac{\begin{array}{c} E;B \vdash y : s \quad E;B \vdash t \\ E;B \vdash s \lesssim t \end{array}}{E;B \vdash^{C} \langle t \rangle y : t} \qquad \text{(T-UPD')}\quad \frac{\begin{array}{c} E;B \vdash z : c\langle\sigma\rangle \quad \mathcal{F}_c(f) = t \\ E;B \vdash s \lhd \sigma_z(t) \quad E;B \vdash y : s \end{array}}{E;B \vdash^{C} z.f = y : \sigma_z(t)}$$

$$\boxed{E;B \vdash^{C}_{\mathcal{B}} b : s}$$

$$\text{(T-CHECK)}\quad \frac{\begin{array}{c} E;B \vdash z : c\langle\sigma\rangle \quad \mathcal{F}_c(f) = t \\ E;B \vdash y : s \quad E;B \vdash s \lhd \sigma_z(t) \end{array}}{E;B \vdash^{C}_{\mathcal{B}} z.f \leftarrow y : \sigma_z(t)} \qquad \text{(T-UPD'')}\quad \frac{\begin{array}{c} E;B \vdash z : c\langle\sigma\rangle \quad \mathcal{F}_c(f) = t \quad E;B \vdash y : s \\ E;B \vdash s \lhd \sigma_z(t) \quad \textbf{specified}(\sigma_z(t)) \end{array}}{E;B \vdash^{C}_{\mathcal{B}} z.f = y : \sigma_z(t)}$$

4.2 Helper relations

If two types are related via $\lesssim$, there is a freedom to choose the run-time semantics of type casts, moving along either $\sim$ or $\leq$ axis. Following the rule (GRAD-SUB), we compute the type $c'\langle\sigma'\rangle$ that is on the same class-level as the target type $c'\langle\sigma''\rangle$ for the upcast. The following lemma justifies this computation:

Lemma 1 (Inversion lemma for $\lesssim$). *If $E;B \vdash t \lesssim t''$, then there exists a type t' such that $E;B \vdash t \leq t'$ and $E;B \vdash t' \sim t''$.*

To construct an "upper-left" mediator type we use an extra helper function $t \uparrow c$ that computes a supertype of the type t at class c.

$$\begin{array}{rl} c\langle\sigma\rangle \uparrow c \triangleq & c\langle\sigma\rangle \\ c'\langle\sigma\rangle \uparrow c \triangleq & d\langle \alpha_j \mapsto \sigma(p_j)_{j\in 1..m}\rangle \uparrow c \\ & \text{where } \texttt{class}\ c'\langle\alpha_{i\in 1..n}\rangle\ \texttt{extends}\ d\langle p_{j\in 1..m}\rangle \text{ and } \texttt{class}\ d\langle\alpha_{j\in 1..m}\rangle \in P. \\ t \uparrow c\langle\_\rangle \triangleq & t \uparrow c. \end{array}$$

In words, the partially defined function $\uparrow$ pulls up the information from the substitution σ of the initial type $c\langle\sigma\rangle$ until it reaches the desired superclass c. If the class hierarchy `Object` is reached without making a match, the function is undefined.

Lemma 2 (Basic properties of $\uparrow$). *For all E, B, t, t',*

1. $(t \uparrow t) = t$
2. $(E;B \vdash t) \wedge (E;B \vdash t') \wedge (t \uparrow t' \neq \bot) \Rightarrow E;B \vdash t \leq (t \uparrow t')$
3. $E;B \vdash t \lesssim t' \Rightarrow E;B \vdash (t \uparrow t') \sim t'$.

The relation $E \vdash t \lhd t'$states that t *satisfies all constraints imposed by known owners of* t'. It is used to detect where type casts should be inserted.

Definition 3 (t is more defined than t').

$$\begin{array}{l} E \vdash t \lhd t' \triangleq E \vdash t \lesssim t' \text{ and } \forall i\ \ q_i \neq ?\ \vee\ p_i \neq q_i \\ \qquad\qquad \text{where } (t \uparrow t') = c\langle p_{i\in 1..n}\rangle \text{ and } t' = c\langle q_{i\in 1..n}\rangle \end{array}$$

If the information about the first owner parameter of the type t of some class field is not known statically, the OAD invariant cannot be guaranteed. In this case a boundary check should be inserted. The predicate **specified**(t) is true iff a type t provides enough static info about ist owners to ensure the OAD invariant preservation.

Definition 4 (t **specifies its owner). specified**$(t) \triangleq p_1 \neq\ ?$*, where* $t = c\langle p_{i\in 1..n}\rangle$

The type rules for type casts and boundary checks are present in Figure 7. For JO$_?^+$ we use different typing relations, namely, $\vdash^{\mathcal{C}}$ and $\vdash^{\mathcal{C}}_{\mathcal{B}}$. These two relations are similar to $\vdash$ for JO$_?$. The purpose of each of them is to ensure the specific safety conditions after the corresponding stage of the translation (type cast and boundary check insertion, respectively). One significant difference is that all the occurrences of $\lesssim$ in the typing of statements are now concentrated in the rule (T-CAST). In the rest of the $\lesssim$-rules are replaced by $\lhd$ (grayed parts). The rule (T-CHECK) ensures the type conformance via $\lhd$, but not the preservation of the OAD invariant: this is postponed until run-time. The rule (T-UPD") is targeted to ensure the OAD invariant.

4.3 Type-directed program translation

We adopt the idea of Siek and Taha [27] to define a type-directed *type cast* insertions and extend it with the *boundary check* insertion relation (Figure 8, relations $\overset{\mathcal{C}}{\rightsquigarrow}$ and $\overset{\mathcal{B}}{\rightsquigarrow}$, respectively). First, type casts are inserted into a program whenever additional information about types needs to be regained. Then the boundary check insertion translation works on the program with inserted casts, so each step of the translation eliminates an aspect of uncertainty caused by incomplete type annotations.

Figure 8 provides the definition of selected rules for the cast insertion relation that specifies the translation. It is written $E \vdash e_1 \overset{\mathcal{C}}{\rightsquigarrow} e_2 : t$ for expressions and holds if, under the assumptions from E, expression e_1 is translated into expression e_2 and the type of e_1 is $\vdash$-determined as t. In the same way it is defined for methods. The rules for classes and a whole program are straightforward and omitted. No cast is inserted if the predicate $\lhd$ holds on types being compared. For conditional insertions we define the helper function $\mathcal{C}$, which uses non-recursive *local decomposition* of an expression e via the context G and optionally inserts type-casts:

$$\begin{aligned} \mathcal{C}_E\langle t_1,t_2\rangle(e) \ &\triangleq\ \textbf{if } (E \vdash t_1 \lhd t_2) \textbf{ then } e \textbf{ else } (\texttt{let } y' = \langle t_2\rangle y \texttt{ in } G[y']) \\ &\quad\ \textbf{where } y' \text{ is fresh, } e = G[y] \\ G &::= [\,] \mid \texttt{let } x = z.m([\,]) \texttt{ in } e \mid \texttt{let } x = (z.f = [\,]) \texttt{ in } e \end{aligned}$$

Boundary check insertion $\overset{\mathcal{B}}{\rightsquigarrow}$ is of the second stage of the whole translation (Figure 8). The translation $\overset{\mathcal{B}}{\rightsquigarrow}$ works on top of the $\vdash^{\mathcal{C}}$-well-typed program. The only type of the statement that can be affected by $\overset{\mathcal{B}}{\rightsquigarrow}$ is a *field update* since it is only one that can possibly break the OAD invariant. The helper function $\mathcal{B}$ is defined to optionally replace plain assignments with boundary-checked field assignments whenever insufficient type information about *primary* owners is provided. For the rest of the statements, expressions and methods, $\overset{\mathcal{B}}{\rightsquigarrow}$ is applied recursively.

$$\begin{aligned} \mathcal{B}\langle t\rangle(b) \ &\triangleq\ \textbf{let } (z.f = y) = b \textbf{ in } (\textbf{if specified}(t) \textbf{ then } b \textbf{ else } z.f \leftarrow y) \\ F &::= [\,] \mid \texttt{let } z = b \texttt{ in } F \end{aligned}$$

Fig. 8 Compilation of $\mathsf{JO}_?$: type-directed translation

$$\boxed{E \vdash e \overset{\mathcal{C}}{\rightsquigarrow} e' : t}$$

(C-UPD)

$$\frac{\begin{array}{c} E \vdash z : c\langle\sigma\rangle \quad \mathcal{F}_c(f) = t \\ E \vdash s \lesssim \sigma_z(t) \quad E \vdash y : s \\ E, x : \mathsf{fill}(x, \sigma_z(t)) \vdash e_1 \overset{\mathcal{C}}{\rightsquigarrow} e_2 : s' \end{array}}{\begin{array}{c} E \vdash \mathtt{let}\ x = (z.f = y)\ \mathtt{in}\ e_1 \overset{\mathcal{C}}{\rightsquigarrow} \\ \mathcal{C}_E\langle s, \sigma_z(t)\rangle(\mathtt{let}\ x = (z.f = y)\ \mathtt{in}\ e_2) : s' \end{array}}$$

(C-CALL)

$$\frac{\begin{array}{c} E \vdash z : c\langle\sigma\rangle \quad \mathcal{MT}_c(m) = (y', t \to t') \quad E \vdash y : s \\ E \vdash s \lesssim \sigma_z(t) \quad \sigma' \equiv \sigma \uplus \{y' \mapsto y\} \\ E, x : \mathsf{fill}(x, \sigma'_z(t')) \vdash e_1 \overset{\mathcal{C}}{\rightsquigarrow} e_2 : s' \end{array}}{\begin{array}{c} E \vdash \mathtt{let}\ x = z.m(y)\ \mathtt{in}\ e_1 \overset{\mathcal{C}}{\rightsquigarrow} \\ \mathcal{C}_E\langle s, \sigma_z(t)\rangle(\mathtt{let}\ x = z.m(y)\ \mathtt{in}\ e_2) : s' \end{array}}$$

$$\boxed{E \vdash t'\ m(t\ y)\{e\} \overset{\mathcal{C}}{\rightsquigarrow} t'\ m(t\ y)\{e'\}}$$

(C-METHOD)

$$\frac{\begin{array}{c} E \vdash e_1 : s \quad E \vdash s \lesssim t' \quad e_2 = F[z] \\ E, y : \mathsf{fill}(y, t) \vdash e_1 \overset{\mathcal{C}}{\rightsquigarrow} e_2 : s \end{array}}{\begin{array}{c} E \vdash t'\ m(t\ y)\{e_1\} \overset{\mathcal{C}}{\rightsquigarrow} \\ t'\ m(t\ y)\{F[\mathcal{C}_E\langle s, t'\rangle(z)]\} \end{array}}$$

$$\boxed{E \vdash e \overset{\mathcal{B}}{\rightsquigarrow} e' : t}$$

(B-UPD)

$$\frac{\begin{array}{c} E \vdash z : c\langle\sigma\rangle \quad \mathcal{F}_c(f) = t \quad E \vdash y : s \\ E \vdash s \lhd \sigma_z(t) \\ E, x : \mathsf{fill}(x, \sigma_z(t)) \vdash e_1 \overset{\mathcal{B}}{\rightsquigarrow} e_2 : s' \end{array}}{\begin{array}{c} E \vdash \mathtt{let}\ x = (z.f = y)\ \mathtt{in}\ e_1 \overset{\mathcal{B}}{\rightsquigarrow} \\ \mathtt{let}\ x = \mathcal{B}\langle\sigma_z(t)\rangle(z.f = y)\ \mathtt{in}\ e_2 : s' \end{array}}$$

Definition 5. *$E \vdash e \rightsquigarrow e'' : t$ iff $E \vdash e \overset{\mathcal{C}}{\rightsquigarrow} e' : t$ and $E \vdash e' \overset{\mathcal{B}}{\rightsquigarrow} e'' : t$ for some $e' \in \mathsf{JO}_?^+$.*

Theorem 1 (Program translation is $\vdash^{\mathcal{C}}_{\mathcal{B}}$-sound.). *$E \vdash e : t$ implies $E \vdash e \rightsquigarrow e' : t$ for some e'. Furthermore, $E \vdash e \rightsquigarrow e' : t$ for some e implies $E \vdash^{\mathcal{C}}_{\mathcal{B}} e' : t$.*

The translation relation $E \vdash e_1 \rightsquigarrow e_2 : t$ can be extended to classes and programs in a straightforward fashion. For instance, we denote $\vdash P_1; e_1 \rightsquigarrow P_2; e_2$ if a program $P_2; e_2$ is obtained from $P_1; e_1$ by the compositional type-directed translation.

5 Operational semantics of $\mathsf{JO}_?^+$

This section provides the definition of dynamic semantics of $\mathsf{JO}_?$. The selected rules of the small-step operational semantics of $\mathsf{JO}_?$ is presented in Figure 9 (the rest of the rules is standard and can be found in the companion technical report [25]). The semantics is in the form of a small-step CEK-like abstract machine with a single-threaded store H, binding environment B and explicit continuations K [14]. A continuation K is, informally, a serialized next step of computation.

$$K ::= \mathbf{mt} \mid \mathbf{call}(x : (t, \sigma), e, B, K) \mid \mathbf{fail}(K)$$

The *empty* continuation **mt** corresponds to the empty control stack which is a case at the beginning and at the correct end of program execution. $\mathbf{call}(x : (t, \sigma), e, B, K)$ describes the discipline of popping the stack when a method ends its execution and its caller's local environment B should be restored with a result assigned to a variable x. Finally, $\mathbf{fail}(K)$ denotes the result of failing casts and boundary checks.

To implement dynamic type casts, we first need a bit of machinery to relate *syntactic* types with *dynamic* types extracted from the object heap during the program execution.

Fig. 9 Small-step operational semantics of $\mathsf{JO}_?^+$ (selected rules)

$$\boxed{\langle H,B,e,K\rangle \Rightarrow \langle H',B',e',K'\rangle}$$

$$\frac{H;B \vdash t \ltimes t' \quad B(y)=\iota \qquad H(\iota)=\langle s,\ldots\rangle \quad \widehat{H} \vdash s \lhd t'}{H;B \vdash \mathsf{cast}(t,y)}\ (\textsc{cast-check})$$

$$\frac{B(y)=\texttt{null} \vee H;B \vdash \mathsf{cast}(t,y) \qquad B' = B[x \mapsto B(y)]}{\langle H,B,\texttt{let}\ x=\langle t\rangle y\ \texttt{in}\ e,K\rangle \Rightarrow \langle H,B',e,K\rangle}\ (\textsc{e-cast1})$$

$$\frac{B(y)\neq\texttt{null} \quad H;B \nvdash \mathsf{cast}(t,y) \quad K\neq\mathbf{fail}(\_) \qquad e=(\texttt{let}\ x=\langle t\rangle y\ \texttt{in}\ e')}{\langle H,B,e,K\rangle \Rightarrow \langle H,B,e,\mathbf{fail}(K)\rangle}\ (\textsc{e-cast2})$$

$$\frac{B(x)=\iota \quad B(y)=\iota' \qquad H(\iota')=\langle c\langle k,\ldots\rangle,\ldots\rangle \quad \widehat{H};\emptyset \vdash \iota \prec k}{H;B \vdash \mathsf{boundary}(x,y)}\ (\textsc{boundary-check})$$

$$\frac{(B(y')=\texttt{null} \vee H;B \vdash \mathsf{boundary}(y,y')) \qquad B(y)=\iota \quad B(y')=v \quad H(\iota)=o \qquad H'=H[\iota \mapsto o[f \mapsto v]] \quad B'=B[x \mapsto v]}{\langle H,B,\texttt{let}\ x=(y.f\leftarrow y')\ \texttt{in}\ e,K\rangle \Rightarrow \langle H',B',e,K\rangle}\ (\textsc{e-boundary1})$$

$$\frac{B(y')\neq\texttt{null} \quad H;B \nvdash \mathsf{boundary}(y,y') \qquad K\neq\mathbf{fail}(\_) \quad e=(\texttt{let}\ x=(y.f\leftarrow y')\ \texttt{in}\ e')}{\langle H,B,e,K\rangle \Rightarrow \langle H,B,e,\mathbf{fail}(K)\rangle}\ (\textsc{e-boundary2})$$

We define a helper relation $H;\ B \vdash t \ltimes t'$ to compute the dynamic type t' corresponding to a static type t in dynamic environments H and B by instantiating owners as follows:

$$\frac{\forall\, i\ \in\ 1..n \quad q_i = \begin{cases} k & \text{if } \begin{cases} p_i = x^{c.j} \\ H(B(x)) = \langle t,\ldots\rangle \\ k = \mathsf{owner}_j(t \uparrow c) \end{cases} & \textit{dependent owner} \\ p_i & \text{if } \mathsf{actual}(p_i) & \textit{run-time owner} \\ B(p_i) & \text{if } \mathsf{defined}(p_i) & \textit{formal owner or variable} \\ ? & \text{otherwise} & \textit{unknown owner} \end{cases}}{H;B\ \vdash\ c\langle p_{i\in 1..n}\rangle \ltimes c\langle q_{i\in 1..n}\rangle}$$

The test $\iota \prec o$ in the rule (BOUNDARY-CHECK) can be performed at run-time by checking whether o is ι or some transitive owner of ι—this information is retained via the flattened heap $\widehat{H}$.

6 Type safety

In this section we sketch the type safety of $\mathsf{JO}_?$ as a corollary of the correctness of the type-guided program translation with respect to program typing and the type safety of the extended language $\mathsf{JO}_?^+$ with type casts and boundary checks. A complete formal treatment with the definition of well-formed run-time states and proofs of theorems is available in the accompanying technical report [25].

Proposition 1 (Compilation and gradual typing). $E \vdash P; e$ *iff* $\exists P', e'.\ E \vdash P; e \leadsto P'; e'$.

The preservation of the OAD invariant relies on three facts: (1) an initial configuration of any program obeys the OAD invariant, (2) the subject reduction theorem guarantees the type preservation for subsequent configurations, and (3) making a step from any well-typed configuration obeying the OAD invariant, preserves the invariant. In the remainder of this section we formalize these statements.

The operational formalism we use is a stack-based abstract machine (continuations form a stack-like structure) with a heap, so we need to separate environments ro provide typing for heap objects and references in stack frames.

$$\begin{array}{ll} \mathcal{E} ::= \emptyset \mid \mathcal{E},\, \iota : c\langle k_{i\in 1..n}\rangle \mid \mathcal{E},\, \iota \prec k & \textbf{\textit{heap environments}} \\ \overline{E} ::= \mathbf{Nil} \mid E \bullet \overline{E} & \textbf{\textit{stack environments}} \end{array}$$

Below in this section we assume that static typing environments E defined in Section 3 contain only term and owner variables in their domain, but not heap locations. A stack environment is *well-formed* if all its constituents are well-formed. The definition of a well-formed run-time state ($\mathcal{E},\overline{E} \Vdash \langle H,B,e,K\rangle$), which is omitted, assumes the expression e to be well-typed ($\mathcal{E},E_0;B \vdash^{C}_{\mathcal{B}} e : t$) and environments $\mathcal{E}$ and $\overline{E}$ well-formed. The last ensures, in particular, that the heap H has no ownership-cycles ($\mathcal{E} \vdash H$).

Lemma 3 (Initial state typing). *$\mathcal{E},E;B \vdash^{C}_{\mathcal{B}} e : t$ iff $\mathcal{E},(E \bullet \mathbf{Nil}) \Vdash \langle H,B,e,\mathbf{mt}\rangle$ for some initial heap H such that $\mathcal{E} \vdash H$.*

Definition 6 (Heap environment extension). *An environment $\mathcal{E}'$ is an extension of $\mathcal{E}$ (written $\mathcal{E}' \gg \mathcal{E}$) if and only if $\mathcal{E} \subseteq \mathcal{E}'$.*

Definition 7 (Stack environment evolution). *We say that a stack environment $\overline{E}$ transforms to a stack environment $\overline{E}'$ (written $\overline{E} \rightsquigarrow \overline{E}'$) if one of the following holds:*

- $\overline{E}' = E' \bullet \overline{E}$ *for some E'* *(method call);*
- $\overline{E}' = (E_0, x : t) \bullet \mathsf{tail}(\overline{E})$ *for some t and $x \notin dom(E_0)$* *(variable assignment);*
- $\overline{E}' = (E_1, x : t) \bullet \mathsf{tail}(\mathsf{tail}(\overline{E}))$ *for some t and $x \notin dom(E_1)$* *(method return).*

Theorem 2 (Subject reduction in $\mathsf{JO}^{+}_{?}$). *If $e \in \mathbf{Expr}$ in $\mathsf{JO}^{+}_{?}$, $\mathcal{S} = \langle H,B,e,K\rangle$, $\mathcal{E},\overline{E} \Vdash \mathcal{S}$ for some well-formed $\mathcal{E},\overline{E}$ and $\mathcal{S} \Rightarrow \mathcal{S}'$ then $\mathcal{E}',\overline{E}' \Vdash \mathcal{S}'$ for some well-formed $\mathcal{E}',\overline{E}'$ such that $\mathcal{E}' \gg \mathcal{E}$ and $\overline{E} \rightsquigarrow \overline{E}'$.*

Theorem 3 ensures that for all well-formed states, if it is possible to make a next step, then the OAD invariant is preserved for the heap component of the resulting state.

Theorem 3 (OAD **preservation in $\mathsf{JO}^{+}_{?}$).** *If $e \in \mathbf{Expr}$ in $\mathsf{JO}^{+}_{?}$, $\mathcal{S} = \langle H,B,e,K\rangle$, $\mathcal{E};\overline{E} \Vdash \mathcal{S}$, $\mathsf{OAD}(H)$ and $\mathcal{S} \Rightarrow \mathcal{S}'$ for some $\mathcal{S}' = \langle H', \_, \_, \_\rangle$ then $\mathsf{OAD}(H')$.*

Definition 8 (Initial state). *Assume $P;e$ to be a program in $\mathsf{JO}^{+}_{?}$, $H = \{\texttt{world} \mapsto \bullet\}$, $B = \{\texttt{this} \mapsto \texttt{world}\}$ is an* initial binding environment. *Then the* initial configuration *of $P;e$ is* $\mathbf{init}(e) = \langle H,B,e,\mathbf{mt}\rangle$.

Following [11], we introduce a class `World` with no owner parameters to represent the object corresponding to the owner of `world`-annotated instances, and for the completeness we need to provide its type. Taking $\mathcal{E} = \{\texttt{world} : \texttt{World}\}$ and $\overline{E} = \{\texttt{this} : \texttt{World}\} \bullet \mathbf{Nil}$, we obtain $\emptyset \vdash^{C}_{\mathcal{B}} P;e \Rightarrow \mathcal{E},\overline{E} \Vdash \mathbf{init}(e)$ by Lemma 3. Theorem 4 ends our chain of safety statements.

Theorem 4 (Static type safety of $\mathsf{JO}_{?}$). *If $\vdash P;e \rightsquigarrow P';e'$ and $\mathbf{init}(e') \Rightarrow^{*} \mathcal{S}$, then one of the following statements holds:*

(a) $\mathcal{S} = \langle H, B, v, \mathbf{mt} \rangle$ *for some* H, B *and* v *(final state);*
(b) $\mathsf{NPE}(\mathcal{S})$ *(null-pointer error);*
(c) $\exists \mathcal{S}' : \mathcal{S} \Rightarrow \mathcal{S}'$ *(progress);*
(d) $\mathcal{S} = \langle H, B, b, \mathbf{fail}(K) \rangle$*, where* $b = \langle t \rangle y$ *or* $b = z.f \leftarrow y$ *for some* H, B, t, y, z, f *and* K *(*OAD *violation attempt).*

Combined Theorems 1, 3 and 4 state that the provided gradual type system ensures that (a) during a compiled program execution no ownership invariant will be violated, and (b) fully-annotated well-typed programs will be executed until the final or null-pointer error state with no ownership invariant violation.

7 Implementation

A prototype compiler for Gradual Ownership Types has been implemented in the JastAdd framework as a small syntactic extension of the JastAddJ compiler for Java [13]. The extension is about 2,600 lines of code, not including tests, blank lines and comments.[3] Although generics were introduced in Java 5, we have chosen Java 1.4 as a host language for the sake of simplicity. Parametric polymorphism is an orthogonal feature to the ownership parametrization, but they can be unified [24].

The type analysis and type-directed translation are implemented as attributes in the framework of reference attribute grammars [13]. The type analysis is built on top of the standard Java type-checking algorithm, which is augmented to handle ownership-parametrized types. The compiler uses several default conventions as well as *manifest ownership* [10] to seamlessly embed the *raw* Java code into an ownership-aware environment. To be parametrized by some owners, a class or an interface requires *all* its super classes and the interfaces it implements to carry ownership parameters. I.e., no casts of ownership-parametrized types to raw types is allowed, since it could lead to breakage of the OAD invariant [24]. The only one exception to this rule is handling of `Object` class. We assume that two `Object` classes exist: one is ownership-parametrized and the other is owned by **`world`** and considered as a special case of the first one. Classes that inherit from parametrized classes or interfaces but do not declare ownership parameters are implicitly assumed to be owned by **`world`**, which is made the owner of the supertypes. The type-directed translation is implemented as a source-to-source transformation by erasing ownership types, augmenting classes with fields for owner parameters and inserting run-time checks into the code of expressions. The compiler might also need to modify code that interacts with owner-parametrized classes, i.e, some libraries might need to be recompiled.

Dependent owners and casts Instead of transforming Java programs into ANF, we operate with dependent owners in terms of *source code locations* corresponding to the expression that computes an owned object. Any expression in the program can hereby give rise to dependent owners, which potentially can be used in further checks. To avoid management of all possible source locations, the compiler runs a simple static analysis to determine which dependent owners might be used in the current context.

[3] A prototype is available from `http://people.cs.kuleuven.be/ilya.sergey/gradual/`

Inner classes and manifest ownership In Java a non-static inner class is nested in the body of another class and contains an implicit reference to its enclosing class (*outer instance*). An instance of such a class can be leaked and referred to through a field by another object *outside* of its outer instance, which, again, breaks the desired invariant. There are multiple suggestions on the problem of interoperation of inner classes and different ownership policies [2, 3]. We make outer instance's ownership parameters legal in the context of an inner class if the programmer passes them to the inner class as owner arguments, i.e., by a sort of closure-conversion. However, most of the time one does not intend an inner class to be parametrized, since it is something for the internal use, but it may be externally accessible. To solve this design problem, we employ *manifest ownership*, a mechanism to allow owned classes without explicit owner type parameters [10]. A manifest class does not have an explicit owner parameter, rather the class's owners are fixed, so all the objects of the class have the same owners.

8 Experience

We evaluated our approach by gradually porting several classes from the Java Collection Framework (Java SDK version 1.4.2) into Gradual Ownership Types. Most traditional collection classes that contain linked data structures implement internal logic to handle their entries in the way it is described in the example in Figure 1. We assume that internal entries should be dominated by their outer collection instances, so they are not supposed to be exposed to the external objects. It makes them a good possible candidate for ownership types and the owners-as-dominators policy. Our intention was to ensure the OAD invariant holds for inner classes such as `Entry` of collection classes such as `LinkedList` and `TreeMap`, without changing existing code, but only by adding annotations. The questions we were trying to answer are:

- How many annotations (i.e., lines of code changed) are needed minimally?
- What is the execution time with minimal annotations?
- How many annotations are needed for full static checking?

The analysed code base consists of 46 source files, comprising about 8,200 lines of code, not including blank lines and comments. The compiler provides hints for easily migrating to ownership types by emitting static error messages and warnings. A static error message is emitted whenever necessary annotations are omitted. A warning message is displayed whenever dynamic casts or boundary checks are inserted.

LinkedList The minimal amount of annotations to ensure the OAD invariant for instances of the inner class `Entry` of `LinkedList` is 17, comprising 7 annotations to the `LinkedList` class itself and 10 in five other classes. Class `Iterator` was owner-parametrized to preserve the OAD invariant, as the inner class `ListItr` has access to entries of the list. The correctly annotated class `ListItr` is defined as follows; the iterator is owned by the instance of `LinkedList` (employing the manifest ownership):

```
class ListItr implements ListIterator<LinkedList.this>
```

We implemented a series of simple benchmarks consisting of multiple list updates and iterations. These reveal that the minimal annotations cause average execution time per update to double. However, the implementation of `LinkedList` allows *full* annotation. By adding 17 extra annotations in the `LinkedList` class (i.e., 34 in total), one can reach zero execution overhead and full static preservation of the OAD invariant.

TreeMap For the best result in terms of performance and the invariant preservation the `TreeMap` class requires 28 annotations, consisting of 26 annotations in the class itself and two extra annotations in the interfaces `Iterator` and `Map` respectively. Because of the static method `buildFromSorted`, which also operates with entries, it is impossible to provide full static ownership guarantee without modifying the original code. The possible solutions would be making the method non-static, or providing an extra final method parameter as an alias for the potential owner. Alternative solution is to use owner-polymorphic methods [9], which are not supported in the current formalism. According to the set of stress benchmarks involving multiple updates and iterations, even in the presence of some non-avoidable casts, the annotated `TreeMap` class exhibits only 30% average execution time overhead per update.

Detected object leaks Our compiler has helped to detect a place in the Collection Framework with the possible "leak" of the inner `Entry` classes with respect to the OAD invariant. The class `ResourceBundleEnumeration` declares a package-protected field of type `Iterator`. Although this field is initialized with the iterator of the `Set` instance in the constructor, it can be reassigned elsewhere in client code, which will lead to an OAD invariant violation. Our compiler generates the code with necessary dynamic checks for updates of this field to ensure the invariant dynamically. However, for the static OAD guarantee a significant refactoring would be required.

9 Discussion

Several design choices were made in our approach to gradual ownership types. This section discusses other alternatives.

Alternative ownership disciplines In our work we used the *owner-as-dominator* discipline as a base for applying the gradual technique. However, most of existing parametric ownership disciplines, such as *multiple ownership* [7] or *ownership domains* [2], can be "gradualized" using similar approach with *no changes* in the part related to type cast insertion. The difference between most of existing disciplines lies in the definition of the heap invariant and relation between owners that should be preserved. In the present work it is ensured by the boundary check, and for any other particular system it might require specific tweaks in the definitions of the consistent-inside relation, **specified** and the runtime semantics of boundary check.

Required annotations The present approach required that ownership parameters be specified (explicitly or via default conventions) at all allocation sites, hence object owners are all known at creation time. Two other possibilities were considered. The first was to annotate field and method types, thereby annotating the interface of the object. This approach unfortunately creates a significant overhead in the implementation, which would require run-time tracking of object aliasing: whenever an object owner becomes known, for example, by assignment into a field whose owners are specified, all other aliases to that object need to be checked for validity. Furthermore, the ownership of objects with the same owner as the assigned object also need to be updated—objects can have the same owner, even if this owner is not known; consider for example, the `Entry` objects in a linked list. The required run-time modifications are likely to introduce too much run-time overhead. The second approach was to allow annotations to occur anywhere in the code. This approach is clearly best suited for programmers, but it clearly also suffers the same problems as annotating just the interface.

Treatment of libraries Our approach essentially assumes that any library code that needs to be owner-aware must be rewritten, but rewriting the library is a significant overhead, the kind which gradual typing aims to avoid. Three alternative approaches are possible. One is to ignore leaks of an object into ownership-unaware code, and assume a weaker ownership invariant that amounts to saying that an object is protected only within code compiled by our compiler. With this more pragmatic approach, library code can more gradually be converted to owner-aware code and trusted library code can 'safely' be ignored. A second alternative is to implement the byte-code instrumentation procedure that inserts the run-time checks to monitor field assignments in the code. The third approach is to perform a static analysis of library (byte)code along the lines of Ma and Foster's work [19] to infer possible ownership annotations.

Boundary checks Boundary checks occur whenever an object is stored in a field of a type with an unknown primary owner of another object in order to preserve the OAD invariant. An alternative interpretation of such a type is that it does not care what the owners are. This would allow expensive boundary checks to be omitted, keeping only dynamic casts, at the expense of a weaker invariant. Such a system may be worth further investigation.

10 Related work

Our work is strongly based on the idea of gradual types by Siek and Taha [26, 27], which has been recently applied to Java-like generics [17] and modular typestate [30]. The notion of *blame control* is known in the context of gradual types to provide better debugging support [29]. Since dependent owners contain information about source code locations, the information from labels makes it easy to track back the flow dependencies and eliminate uncertainty by adding extra ownership annotations. This makes dependent owners similar to *blame labels*. The idea of combining static and dynamic type checking is also close to the work of Flanagan on *hybrid types* [15]. Hybrid types may contain refinements in the form arbitrary predicates on underlying data. The type checker attempts to satisfy the predicates statically using a theorem prover.

Gordon and Noble in the work on dynamic ownership introduce ConstraintedJava, a scripting language that provides dynamic ownership checking [16]. The authors suggest a dynamic ownership structure consisting of an owner pointer in every object. The semantics of the language relies on a message-passing protocol with a specific kind of monitoring, similar to our boundary checks.

Existential ownership types [18] offer variant subtyping of owners based on existential quantification [6]. This approach allows *owner-polymorphic methods* to be elegantly implemented and it distinguishes objects with different and equal *unknown* owners. Existential quantification also helps to implement effective run-time downcasts in the presence of ownership types: a subtype's inferred owners are treated as existentially quantified [32]. The key difference between these approaches and ours is that existential ownership expresses *don't care* whereas gradual types express *don't know* concerning the unknown owners.

Algorithms for *ownership inference* address a similar problem to ours: take a raw program and produce reasonable ownership annotations. The pioneering work on *dynamic* ownership types' inference is Wren's master's thesis [31]. The work provides a graph-theoretical background for run-time inference. The author formulates the system of equations to assign annotations to particular object allocation sites, based on an object graph's evolution history. However, no proof of correctness of these equations is provided. Milanova and Vitek [20] present a static analysis to infer ownership annotations for the OAD invariant. The analysis is based on the context-insensitive points-to analysis. A more general points-to analysis-based algorithm to infer ownership and uniqueness is presented by Ma and Foster [19] via constraint-based points-to analysis. The collected information about encapsulation properties is not however mapped to a type system. Dietl et al. [12] present a static analysis to infer *Universe Types*, a light-weight version of ownership types, according to a set of generated constraints. Constraints of the type system are encoded as a boolean satisfiability problem.

11 Conclusion

Introducing ownership types into real-life programs is a long-standing problem. The main causes are the verbosity of the formalism and its rigidity for some applications. In this work we applied the notion of gradual types to ownership type systems and the owners-as-dominators invariant for a Java-like language to seamlessly combine static and dynamic invariant checks. The developed framework has been formalized and proved to be correct [25]. We implemented Gradual Ownership Types as an extension of an existing Java compiler and evaluated it on a well-studied codebase.

Acknowledgements

We wish to acknowledge the detailed comments of Dominique Devriese, Frank Piessens and Jan Midtgaard. We are also grateful to Sophia Drossopoulou and José Proença for proof-reading an early draft version of the paper. Finally, we thank the anonymous reviewers of ESOP '12 for their feedback, which helped to make the motivation clearer.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.
2. J. Aldrich and C. Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *ECOOP 2004*, volume 3086 of *LNCS*, pages 1–25. Springer-Verlag, 2004.
3. C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *POPL '03*, pages 213–223. ACM, 2003.
4. C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *OOPSLA '01*, pages 56–69. ACM, 2001.

5. C. Boyapati, A. Salcianu, W. Beebee, Jr., and M. Rinard. Ownership types for safe region-based memory management in real-time Java. In *PLDI '03*, pages 324–337. ACM, 2003.
6. N. Cameron and S. Drossopoulou. Existential quantification for variant ownership. In *ESOP 2009*, volume 5653 of *LNCS*, pages 128–142. Springer-Verlag, 2009.
7. N. R. Cameron, S. Drossopoulou, J. Noble, and M. J. Smith. Multiple ownership. In *OOPSLA '07*, pages 441–460. ACM, 2007.
8. D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA '02*, pages 292–310. ACM, 2002.
9. D. Clarke and T. Wrigstad. External uniqueness is unique enough. In *ECOOP 2003*, volume 2743 of *LNCS*, pages 176–200. Springer-Verlag, 2003.
10. D. G. Clarke. *Object ownership and containment*. PhD thesis, University of New South Wales, New South Wales, Australia, 2001.
11. D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA '98*, pages 48–64. ACM, 1998.
12. W. Dietl, M. D. Ernst, and P. Müller. Tunable Static Inference for Generic Universe Types. In *ECOOP 2011*, volume 4609 of *LNCS*, pages 333–357. Springer-Verlag, 2011.
13. T. Ekman and G. Hedin. The JastAdd extensible Java compiler. In *OOPSLA '07*, pages 1–18. ACM, 2007.
14. M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex, 1st edition*. The MIT Press, August 2009.
15. C. Flanagan. Hybrid type checking. In *POPL '06*, pages 245–256. ACM, 2006.
16. D. Gordon and J. Noble. Dynamic ownership in a dynamic language. In *DLS '07*, pages 41–52. ACM, 2007.
17. L. Ina and A. Igarashi. Gradual typing for generics. In *OOPSLA '11*, pages 609–624. ACM, 2011.
18. Y. Lu and J. Potter. On ownership and accessibility. In *ECOOP 2006*, volume 4067 of *LNCS*, pages 99–123. Springer-Verlag, 2006.
19. K.-K. Ma and J. S. Foster. Inferring aliasing and encapsulation properties for Java. In *OOPSLA '07*, pages 321–336. ACM, 2007.
20. A. Milanova and J. Vitek. Static dominance inference. In *TOOLS 2011*, volume 6705 of *LNCS*, pages 211–227. Springer-Verlag, 2011.
21. S. E. Moelius III and A. L. Souter. An object ownership inference algorithm and its applications. In *MASPLAS '04*, 2004.
22. M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A Nominal Theory of Objects with Dependent Types. In *ECOOP 2003*, volume 2743 of *LNCS*, pages 201–224. Springer, 2003.
23. B. C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.
24. A. Potanin, J. Noble, D. Clarke, and R. Biddle. Generic ownership for generic Java. In *OOPSLA '06*, pages 311–324. ACM, 2006.
25. I. Sergey and D. Clarke. Gradual Ownership Types. Technical Report Report CW 613, Katholieke Universiteit Leuven, December 2011.
26. J. Siek and W. Taha. Gradual typing for functional languages. In *Scheme '06*, 2006.
27. J. Siek and W. Taha. Gradual typing for objects. In *ECOOP 2007*, volume 4609 of *LNCS*, pages 2–27. Springer-Verlag, 2007.
28. J. Vitek and B. Bokowski. Confined types. In *OOPSLA '99*, pages 82–96. ACM, 1999.
29. P. Wadler and R. B. Findler. Well-Typed Programs Can't Be Blamed. In *ESOP 2009*, volume 5653 of *LNCS*, pages 1–16. Springer-Verlag, 2009.
30. R. Wolff, R. Garcia, É. Tanter, and J. Aldrich. Gradual typestate. In *ECOOP 2011*, volume 4609 of *LNCS*, pages 459–483. Springer-Verlag, 2011.
31. A. Wren. Inferring ownership. Master's thesis, Imperial College London, UK, June 2003.
32. T. Wrigstad and D. Clarke. Existential owners for ownership types. *Journal of Object Technology*, 6(4), 2007.