Towards gradual ownership types

Ilya Sergey Dave Clarke

IBBT-DistriNet
Department of Computer Science
Katholieke Universiteit Leuven, Heverlee, Belgium
`first.last@cs.kuleuven.be`

Abstract

In this work we present a formal system to annotate programs with ownership types in a lightweight way, allowing only partial information about object owners. We adapt the system of Clarke and Drossopoulou to include gradual types of Siek and Taha. The resulting framework allows one to annotate programs incrementally with ownership types. For fully annotated programs the developed formalism provides a static guarantee of the desired encapsulation invariant, whereas for partially annotated programs necessary dynamic checks are inserted by the compiler.

1. Introduction

Ownership types provide a declarative way to statically enforce the notion of encapsulation in object-oriented programs. A type system that enables control over *object ownership* in Java-like languages is usually plugged into the traditional syntax as a set of type annotations. Variants of ownership types allow a program to enjoy such computational properties as data race-freedom [7], disjointness of effects [11], various confinement properties [33] and effective memory management [8]. It also enables modular reasoning using knowledge about aliasing [28].

However, the verbosity of ownership types seems to be a big obstacle on the way of introducing them into mainstream programming. Indeed, in order to enable the compiler to reason about aliasing, a programmer should provide full annotations for a program or they should be inferred. Unlike traditional type schemes à la Hindley-Milner, ownership annotations are mostly *design-driven*. There is not much use in the inference of ownership types, when no extra annotations is provided since the correct ownership typing always exists [17]. But of course some annotations can be inferred if the necessary amount of information is provided to indicate the intent of the programmer. It would therefore be useful to have a technique that helps to migrate smoothly from an ownership-free program to an ownership-annotated one. For instance, one might be interested in the run-time behaviour of objects with respect to some regions: an object cannot *leave* its owner's scope, i.e., it cannot be assigned to fields of other objects outside of its owner's scope (this is so-called *weak* owners-as-dominators (OAD) invariant [29]). The question is: *must all types be annotated to ensure this behaviour?*

Intuitively, one does not need to provide ownership annotations for *all* types in the program in order to ensure the owners-as-dominators invariant *dynamically* holds: we need only small amount of hints to maintain the invariant. One can consider this as *dynamic checking* for ownership structures. In addition, it would be nice to reason about the invariant *statically* and even eliminate some dynamic checks if sufficient amount of information is provided via annotations. This is the essence of *gradual types*: both static and dynamic checking are possible in the same language by allowing the programmer to control whether some parts of the program can be checked statically or at run-time by adding or removing type annotations [31, 32].

In this paper we explore the idea of applying gradual types to ownership typings and the ownership invariant. Our work is in the spirit of BabyJ by Anderson and Drossopoulou [5]: complete annotation implies the static guarantee of the desired invariant. To this end, we introduce the notion of *gradual ownership types* and the corresponding *consistent-subtyping* relation to reason about ownership structures statically in the presence of only partially-annotated types.

2. Motivation

The possibility of omitting some annotations enables the programmer to write the code that will not violate the invariant, but will be checked *dynamically* if it is impossible to prove the correctness at compile-time. This is useful for the incremental migrating to ownership-aware code. By providing a minimal amount of annotations, one can test the program and figure out if the provided annotations are safe or not.

In the case of ownership types, there are several ways to define which annotations are considered to be "primary", i.e., defining the strategy of dynamic checks and, thus, should be provided by the programmer. In this section we discuss two possible strategies. Section 2.1 describes an approach "instances-first", when the minimal amount of annotations should be provided at allocations sites to define the ownership restrictions. This approach will be further formalized in the paper. In Section 2.2 we describe an alternative design strategy "references-first" that allows annotations to be omitted even on allocation sites.

2.1 "Instances-first" design of gradual types

Informally, the weak owners-as-dominators invariant [29] is as follows. *Given an object o and its* owner*, some other object* α*, then every path in the object graph of a program from* roots (i.e., objects with no input edges) *along objects' field references that ends in o, passes "through"* α. This informal definition does not say a word about annotations of fields, variables etc. Thus, all we need to know to explore the run-time owners-as-dominators property is what each object's owner is. This is a necessary bit of information that should be provided by the programmer. Any field or variable annotations *do not contribute* somehow to this information, and gradual types allow them to be omitted.

We present the series of examples to make this point clearer. Figure 1 gives a motivating example taken from Clarke and Drossopoulou [11]. Ownership annotations are attached to types in angle brackets `<...>` similar to generics. One can notice that only annotations emphasized by grayed boxes are necessary. First, we cannot omit owner parameters in declarations of classes, such as **`class`** `List<owner,data>`, since they define the discipline to name each class' owners and bind them to the owners declared in a

```
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 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 Iterator <o,d> {
  Link<o,d> current;
  Iterator(Link<o,d> first) {
    current = first;
  }
  void next() { current = current.next; }
  Data<d> elem() { return current.data; }
  boolean done() { return (current == null); }
}
```

Figure 1. A list and its iterator: example code with optional and necessary ownership annotations in the "instances-first" gradual ownership types discipline

superclass. Eventually, this defines the runtime ownership mapping from the class' parameters to the owners of the instances allocated within methods of a class. Second, we should keep the ownership arguments at object allocation sites, such as, for instance, `new Link<this, data>()`, because they describe actual ownership structure of particular objects.

To illustrate the idea of gradual ownership typings, consider the following code fragment, which violates the ownership invariant:

```
List<?, ?> list = new List<p, world>();
Iterator<?> iter = list.makeIterator();
ElementFactory<?> factory = new ElementFactory<p>();
factory.iterator = iter; // invariant violation
```

We intentionally left only the minimal necessary amount ownership annotations to show how the invariant imposed by them can be violated at run-time. All non-mandatory annotations are replaced by *unknown owners* "?". Later in the paper, types with no annotations are just syntactic sugar for types with all ownership annotations unknown, e.g., `List` ≡ `List<?,?>`. The object of type `Iterator` created in and returned by the method `makeIterator` of the object `list` has the object `list` as its owner. It is permitted to use `iter` within the stack frame where its owner `list` is available. This discipline is referred as *dynamic aliasing* [11]. However, the invariant is violated the moment `iter` is assigned to the field `iterator` of the object `factory`: the latter has some object `p` as its owner, but not `list` or `iter`, which would be valid.

The described violation could be detected statically in the presence of *full* ownership annotations. Otherwise the invariant violation should be prevented by a run-time check.

2.2 "References-first" design of gradual types

One could imagine the idea of a factory object that creates objects not belonging to anyone. Later these objects might find their owners, by being assigned to some other object's fields with explicitly

```
class ObjectFactory {
  static Object<?> createUnadopted() {
    return new Object<?>() {...}
  }
}
class Adopter<p> {
  Object<p> adoptedChild;
  void adopt() {
    ObjectFactory factory = new ObjectFactory();
    Object<?> unadopted = factory.createUnadopted();
    adoptedChild = unadopted;
  }
}
```

Figure 2. Objects with non-specified owners as an example of the "references-first" discipline of gradual ownership types

specified owners. The fragment of code in Figure 2 provides some intuition about this idea. An instance of the class `ObjectFactory` creates an object with an unspecified owner. In the method `adopt()` the instance of the class `Adopter` parametrized by an owner `p` assigns this object to its field, thus, specifying `unadopted`'s owner to `p`. The "references-first" design strategy allows even more relaxed ownership annotations: ownership annotations can be omitted on both allocation sites and references. In the most relaxed case, only ownership class parameters should be declared, but this also could be implemented by introducing some default agreements (e.g., each class has only one ownership parameter). However, this approach requires a run-time mechanism to monitor assignments and argument passing in order to *fix* concrete owners in the run-time structure of objects.

We keep the discussion of the various design choices for the future work and dedicate the rest of this paper to the "instances-first" design strategy.

3. Combining Gradual Typing and Ownership

In this section we provide the definition of the syntax of $\mathsf{JO}_?$, and the static semantics of type checking in the presence of unknown owners. Following the road of gradual types it seems to be a natural idea to introduce a notion of the special *unknown owner* "?". Unlike the static approach of quantifying ownership types with wildcards [9], program points annotated with "?" in a gradually-typed language help to intentionally omit the pieces information that should be checked at run-time via checks inserted by the compiler.

3.1 Syntax

In order to formalize the notion of gradual ownership types, we define the language $\mathsf{JO}_?$ as an extension of the system JOE_1 of Clarke and Drossopoulou [11] with the *unknown owner* "?". Effect annotations and sub-effecting from JOE_1 are omitted as irrelevant to the topic and for the sake of simplicity. The unknown owner "?" can appear in ownership annotations just like a normal owner.

In $\mathsf{JO}_?$, we distinguish between *concrete* and *abstract* owners. Concrete owners are represented syntactically by owner and term variables, *dependent* owners and *actual* owners such as **world** and heap locations. The unknown owner "?" is considered *abstract* in the sense that it is present as an owner only syntactically in the static semantics of the language and does not provide any information about the corresponding run-time owner. $z^{c.i}$ denotes the i-th dependent owner of the object referred by the term variable z, 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. The purpose of dependent owners is similar to existential owners: they keep an information about

c	$\in$	**ClassName**		
$class$	$\in$	**Class**	$::=$	$\texttt{class}\ c\langle \alpha_{i\in 1..n}\rangle$ $\texttt{extends}\ c'\langle r_{i\in 1..n'}\rangle$ $\{fd_{j\in 1..m}\ \ meth_{k\in 1..p}\}$
f	$\in$	**FieldName**		
fd	$\in$	**Field**	$::=$	$t\ f$
m	$\in$	**MethodName**		
$meth$	$\in$	**Method**	$::=$	$t\ m(t\ x)\ \{e\}$
e	$\in$	**Expr**	$::=$	$z \mid \texttt{let}\ x = b\ \texttt{in}\ e$
x,y	$\in$	**TermVar**		
z	$\in$	**Var**	$::=$	$\texttt{this} \mid x$
b	$\in$	**Comp**	$::=$	$z.f \mid z.f = z \mid z.m(z) \mid \texttt{new}\ c\langle r_{i\in 1..n}\rangle \mid \texttt{null}$
α	$\in$	**OwnerVar**		
r	$\in$	**ConcreteOwner**	$::=$	$z \mid \alpha \mid k$
u	$\in$	**KnownOwner**	$::=$	$r \mid z^{c.i}$
p,q	$\in$	**Owner**	$::=$	$u \mid ?$
k	$\in$	**ActualOwner**	$::=$	$\texttt{world} \mid \ldots$
s	$\in$	**SourceType**	$::=$	$c\langle u_{i\in 1..n}\rangle$
t	$\in$	**Type**	$::=$	$c\langle p_{i\in 1..n}\rangle \mid s$
P	$\in$	**Program**	$::=$	$\{class_j \mid j \in 1..m\}; e$
ι	$\in$	**Location**		
w	$\in$	**VarOrLoc**	$::=$	$z \mid \iota$
v	$\in$	**Value**	$::=$	$\iota \mid \texttt{null}$
κ	$\in$	**ActualOwner**	$::=$	$\ldots \mid \iota$

Figure 3. Static and dynamic aspects of $\mathsf{JO}_?$

the origin of some owner arguments without knowledge about the nature of the owners. The following code fragment provide some intuition about dependent owners:

```
class E <owner, q> {
  D<?> d = new D<q>();
}
class D<owner> {
  E<owner> e;
  void use(D<owner> arg) {...}
  void exploit(E<owner> arg) {
    this.e = arg;
  }
}
E<p> e = new E<p, world>();
D<?> d = e.d; // implicitly, d: D<d^{D.1}>
d.use(d); // coarsening, but no type cast required
d.exploit(e); // dynamic type cast required
```

Without dependent owners we would lose precision when checking types.

Figure 3 provide the definition of the full syntax of $\mathsf{JO}_?$. Programs in our calculus are in the a-normal form, i.e., all intermediate expressions are named. Thus, there is no need in plain variable assignments such as $z = y$. Also, this allows local variables to be used as owners, as long as they do not escape the scope of a local stack frame.

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 *within* relation $\prec^*$.

$$E \;\in\; \textbf{Env} \;::=\; \emptyset \mid E, w : s \mid E, r \prec^* r'$$

The dynamic semantics is defined in Section 4 in terms of an explicit binding of free variables, rather than via substitution. The presence of binding environment in the typing judgements does not affect the static semantics of $\mathsf{JO}_?$, but we will need it to establish equalities between typing environments and dynamic bindings doing the proof of the type preservation theorem. To avoid duplicating work, we include a binding list in the assumption set of most judgements. The bindings B map context variables to actual contexts and variables to values:

$$B \;\in\; \textbf{Binding} \;::=\; \emptyset \mid B, \alpha = k \mid B, z = v$$

The relation of a *well-formed* pair $E;B \vdash \diamond$ is described via rules via rules (IN-BIND1), (IN-BIND2) and (IN-BIND3) in Figure 8. Note, owners in types are defined modulo equality in the binding list. To keep the presentation tractable, we omit explicit mentioning of the rules dealing with such equalities. We also use three helper functions to operate with types and classes: owner, owners and arity.

$$\begin{array}{rcl} \mathsf{owner}(c\langle\rangle) & \triangleq & \texttt{world} \\ \mathsf{owner}(c\langle p_{i\in 1..n}\rangle) & \triangleq & p_1,\ \text{where } n > 0 \\ \mathsf{owners}(c\langle p_{i\in 1..n}\rangle) & \triangleq & p_{i\in 1..n} \\ \mathsf{arity}(c) & \triangleq & n,\ \text{where } \texttt{class}\ c\langle\alpha_{i\in 1..n}\rangle\{\ldots\} \in P \end{array}$$

The definition of well-formed owners in typing environment E is shown in Figure 4 ($E;B \vdash p$). The rules (OWN-DEPENDENT) and (OWN-UNKNOWN) are specific for the gradual type system. (OWN-DEPENDENT) ensures that the dependent owner is valid in any environment, even if the corresponding variable is not present in the environment. The index i may not exceed the arity of the class c. This might seem odd, but, in fact, this relaxation still does not allow dependent owners to "escape" the local context. The rule (OWN-UNKNOWN) states that the unknown owner is valid in any environment.

3.3 Owners ordering

The type system tracks two orders: the order of owners ($\prec^*$) and the ordering of types imposed by the subtyping relation ($\leq$). The definition of owner orderings distinguishes between *known* owners, such as term variables, locations, dependent owners and owner variables, and *unknown* owners. In Figure 4 the traditional order relation is defined for *known* owners only, and a new one, *consistent-subowner* ($E;B \vdash p \preceq p'$) is handled via three new rules: (SUBOWN-LET), (SUBOWN-RIGHT) and (SUBOWN-INCLUDE). The notion of owners ordering is tightly bound with the notion of the *well-formed typing environment*:

Definition 3.1 (Well-formed typing environment)**.** A typing environment E is *well-formed* if $\prec^*$ is a *partial order* on $\{r \mid r \in \mathrm{dom}(E)\}$.

Unlike the plain ordering of owners, the owner consistency relation is *not* transitive, so $E;B \vdash q \preceq ?$ and $E;B \vdash ? \preceq p$ do not imply $E;B \vdash q \preceq p$ unless $q = p = ?$. The rule (SUBOWN-INCLUDE) states that the subowner relation $\prec^*$ on concrete owners implies consistent-subowner relation. Types can be constructed from any class using any owner from a 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. This is a relaxed requirement to ensure that the owners-as-dominators property is maintained. We can statically ensure that all *known* owners do not break the property and the rest (involving "?") will be postponed to run-time checks by the compiler.

3.4 Type consistency and subtyping

We would like to compare types containing an unknown owner "?" as an owner parameter taking the subtyping relation into the account. Every type in the system can be thought as a pair $c\langle\sigma\rangle$, where c is a class name corresponding to some definition in a class table and $\sigma = \{\alpha_i \mapsto p_i \mid i \in 1..\mathsf{arity}(c)\}$ is a substitution from the formal owner parameters of class c to owner arguments, either concrete or abstract. Since two types sharing the same class name can differ in the owner substitutions, 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 type consistency relation answers the question: *which pairs of static types could possibly correspond to comparable run-time types?*

The definition of the subtyping is standard and is taken from the JOE_1 calculus (Figure 5). In order to eliminate non-determinacy from the type-checking algorithms one should design a relation which will combine two kinds of subsumption of types: type consistency and subtyping. This relation is used then in type rules whenever an implicit upcast is necessary [30]. Siek and Taha suggest a way to design such *consistent-subtyping* relation for the calculus $\mathbf{Ob}_{<:}$ of Abadi and Cardelli [1]. However, the proposed approach handles *structural* rather than *nominal* subtyping, which is typical for Java-like languages and is the norm in mainstream object-oriented programming languages. Therefore, one additional contribution of this work is integrating a *consistent-subtyping* relation into a type system with nominal subtyping and parametrized types.

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:

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

This intuition is formalized via the rule (GRAD-SUB) in Figure 5.

3.5 The type rules

Typing rules for expressions are standard (Figure 6). Following the standard approach, instead of using subsumption, we use the consistent subtype relation where it is necessary [30]. $m \uplus m'$ 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. The helper function fill (Definition 3.2) converts declared types with *unknown* owners to types with *dependent* owners to track owner dependencies.

Definition 3.2 (Type conversion)**.**

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

The rules for class typing remain unchanged from the original work on JOE_1. The class `Object` is located on the top of class hierarchy and it has only one owner parameter. A program is a set of classes and an expression. It is well-formed if its constituent classes are well-formed and an expression is well-typed. The system we consider has only trivial order on owners. More expressive possibilities exist, for example, by declaring the expected relationship between owner parameters of a class [13].

4. Operational semantics of $\mathsf{JO}_?$

This section provides the definition of dynamic semantics of $\mathsf{JO}_?$. The small-step operational semantics of $\mathsf{JO}_?$ is presented in Figure 7. 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 [18]. We have chosen this model since it can be easily extended with new types of computations and expressions.

$$\begin{array}{rcllcl} H & \in & \textbf{Heap} & \triangleq & \textbf{Location} \rightharpoonup \textbf{Object} \\ o & \in & \textbf{Object} & ::= & \langle c\langle k_{i\in 1..n}\rangle, [f \mapsto v_{f\in\mathrm{dom}(\mathcal{F}_c)}]\rangle \\ K & \in & \textbf{Continuation} & ::= & \mathbf{mt} \mid \mathbf{call}(x : [\![t,\sigma]\!], e, B, K) \end{array}$$

An object is represented by its runtime ownership structure (i.e., its class name and actual ownership parameters: either `world` or some non-null heap locations). A heap H is a partially defined map from locations to objects. Finally, a continuation K is informally a serialized "next step of computation". The *empty* continuation $\mathbf{mt}$ corresponds to the empty control stack which is a case at the beginning and at the correct and of the program execution. $\mathbf{call}(x : [\![t,\sigma]\!], e, B, K)$ describes the discipline of "popping the stack" when an actual method ends its execution and its caller's local environment B should be restored. A variable x to which the result of the method will be assigned is annotated with the original return type t of the called method and the local substitution σ. These annotations can be obtained during the type-checking phase via the conclusion of the rule (T-CALL). The annotations do not affect the dynamic semantics and are used only for the soundness proof. The following proposition states the equivalence between the semantics of JOE_1 and $\mathsf{JO}_?$ *by construction* [16].

Proposition 4.1 (Equivalence of Semantics)**.** $\forall\, H,\, H',\, B,\, v,\, e \in \mathsf{JOE}_1 \cap \mathsf{JO}_?$,

$$\langle H;B;e\rangle \Longrightarrow \langle H';v\rangle \iff \exists B'.\langle H,B,e,\mathbf{mt}\rangle \Rightarrow^* \langle H',B',v,\mathbf{mt}\rangle.$$

5. Compilation Semantics of $\mathsf{JO}_?$

This section describes the procedure of type-based compilation of programs in $\mathsf{JO}_?$ into an extended language with run-time checks.

5.1 Intuition: *boundary checks* and *type casts*

In the original work on gradual types, Siek and Taha introduce type casts in order to ensure that the run-time structure of a λ-expression corresponds to the type imposed either by type specification or type-checking rules [32]. Later, in the work on gradual typing for objects, the type casts are imposed by partially provided type annotations and the structure of objects [31]. To get some intuition on run-time checks in the case of ownership types, we first recall the owners-as-dominators invariant as it is defined by Östlund and Wrigstad [29]. To state the invariant we need a definition of a heap flattening.

Definition 5.1 (Heap flattening)**.**

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

The notation $\widehat{H}$ is used also to *flatten* a heap H into a typing environment.

Definition 5.2 (Owners-as-Dominators Invariant)**.** $\mathsf{OAD}(H) \triangleq$ for all locations ι, ι' and actual owners o,

$$\left.\begin{array}{c} H(\iota) = \langle c\langle k_{i\in 1..n}\rangle, [f \mapsto v_{f\in\mathrm{dom}(\mathcal{F}_c)}]\rangle \\ f_i \mapsto \iota' \\ \mathsf{owner}(H(\iota')) = o \end{array}\right\} \quad\Rightarrow\quad \widehat{H};\emptyset \vdash \iota \prec^* o$$

Now it is time to look back to the type checking rules and describe in detail how they relax the ownership invariant statically. Informally, the work of the relaxed invariant check is contained in the rules (EXP-CALL), (EXP-METHOD) and (EXP-UPDATE) due to the presence of the relation $\lesssim$ in the premises. But 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. We will refer to this specific kind of dynamic checks as *boundary checks*.

Method calls and returns *cannot* violate the ownership invariant, but they allow some dynamic boundary checks to be avoided thanks to *type refinement*. It is important to notice that *type casts* are needed whenever additional information about types needs to be regained. In general such places occur whenever a value is assigned to a local variable or field or passed to or returned from a method. In the system we consider only method calls and returns and assignments to fields, as `let`-expressions are not annotated with types and the types are inferred.

Essentially, *type casts* and *boundary checks* are two orthogonal procedures. The former are standard for gradual type systems and perform a postponed check that the run-time structure of a datatype corresponds to the programmer's expectations. The latter are particular to systems with ownership types: they postpone the check that the program does not violate the OAD invariant. Section 5.5 provides a *two-pass type-directed* algorithm to sequentially insert both of these types of checks into the program code.

5.2 Syntax of checks

The syntax is extended for dynamic type casts and boundary checks. First, we introduce two types of statements. Second, we define a new sort of continuation $\mathbf{fail}(k)$ to denote the result of failing casts and boundary checks.

$$\begin{array}{rcllcl} b & \in & \textbf{Comp} & ::= & \ldots \mid \langle t \rangle z \mid z.f \leftarrow z \\ K & \in & \textbf{Continuation} & ::= & \ldots \mid \mathbf{fail}(K) \end{array}$$

We refer to the extended version of $\mathsf{JO}_?$ as $\mathsf{JO}_?^+$. Casts and checks are not supposed to be put in by the programmer. They are inserted instead by the cast and check insertion procedures described in Section 5.5. The statement $\langle t \rangle z$ ensures that the run-time type of an object referred to by z *matches* the type t. The statement $x.f \leftarrow y$ imposes the check that a field reference from x to y via the field f would not violate the ownership invariant and performs the field update.

5.3 Type coercion

If two types are related via $\lesssim$, there is a freedom to choose the run-time semantics of type casts. In the original work on gradual types for objects, the authors chose to check the *subtyping* at run-time via type casts (i.e., move along the y-axis on the picture from Section 3.4). More concretely, given $t \lesssim t''$, an intermediate type t' such that $t \sim t'$ is built *statically*. So, only the subsumption $t' \leq t''$ is to be checked at run-time, and this is implemented via the mechanism of type casts. In our case the definition of $\lesssim$ already gives us an algorithm to compute an "upper-left mediator". Following the rule (GRAD-SUB), we compute the type $c'\langle\sigma'\rangle$ that is on one class-level with the target type $c'\langle\sigma''\rangle$ for the upcast.

Lemma 5.3 (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 $E;B \vdash t'' \sim t'$.*

The dynamic *type cast* is then to be used to check the *run-time* consistency of two types with gradual owners. To construct an "upper-left" mediator type we need an extra helper function. The function $t \uparrow c$ computes a basic type of the type t sought for the definition of the consistent subtyping with resect to the supertype of t at class c.

Definition 5.4 (Type coercion ($\uparrow$)).

$$\begin{array}{rcl} c\langle\sigma\rangle \uparrow c & \triangleq & c\langle\sigma\rangle \\ c'\langle\sigma\rangle \uparrow c & \triangleq & d\langle \alpha_j \mapsto \sigma(p_j)_{j\in 1..m}\rangle \uparrow c \\ & & \text{where } \texttt{class } c'\langle \alpha_{i\in 1..n}\rangle \\ & & \qquad\quad \texttt{extends } d\langle p_{j\in 1..m}\rangle\{\ldots\} \\ & & \text{and } \quad \texttt{class } d\langle \alpha_{j\in 1..m}\rangle\{\ldots\}. \end{array}$$

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

$$t \uparrow c\langle\sigma\rangle \quad \triangleq \quad t \uparrow c.$$

$$\begin{tikzcd} (t \uparrow t') \arrow[r, "\sim"] & t' \\ t \arrow[u, "\leq"] \arrow[ur, "\lesssim"'] & \end{tikzcd}$$

The following lemma states the basic properties of $\uparrow$.

Lemma 5.5 (Basic properties of $\uparrow$). *For all E, B, t, t',*

1. $(t \uparrow t) = t$
2. $\left.\begin{array}{c} E;B \vdash t \\ E;B \vdash t' \\ (t \uparrow t') \neq \bot \end{array}\right\}$ *implies* $E;B \vdash t \leq (t \uparrow t')$
3. $E;B \vdash t \lesssim t'$ *implies* $E;B \vdash (t \uparrow t') \sim t'$

5.4 "More defined than" and "specified" relations

Our next step is to figure out what particular checks should be performed at run-time. Actually, all relations between *known* owners might be already inferred at the type checking stage and, thus, statically checked via the rules for owner ordering as it was discussed in Section 5.1. Since the consistency on owners is symmetric, the uncertainty can be caused by the lack of information about owners both from the side of a *source* and *target* type. By *source* types we mean the *inferred* types of call arguments, method results and values to be assigned to fields. *Target* types are those to which the cast is made, namely, declared types of parameters, method returns and fields.

The relation $\lhd$ is is designed to answer the question: *does the left operand satisfy all constraints imposed by known owners of the right operand?*

Definition 5.6 (t is more defined than t').

$$\begin{array}{rcl} t \lhd t' & \triangleq & t \lesssim t' \text{ and } \{i \mid q_i \neq ? \wedge p_i \neq q_i\} = \emptyset \\ & & \text{where } (t \uparrow t') = c\langle p_{i\in 1..n}\rangle \text{ and } t' = c\langle q_{i\in 1..n}\rangle \end{array}$$

If for some concrete owner q_i of the right operand, the corresponding owner of the left operand p_i is either *unknown* or is some *dependent* owner, the function returns false. We use $\lhd$ to detect where type casts should be inserted.

The lack of information in the target type can lead to the violation of the ownership invariant. Such places in code are candidates for boundary check insertion. We use the predicate **specified** to answer the question *does the target type provide enough static information about its owners to ensure the invariant preservation?*

Definition 5.7 (t specifies its owner). $\mathbf{specified}(t) \triangleq p_1 \neq ?$, where $t = c\langle p_{i\in 1..n}\rangle$

If the information about the first owner parameter of the source type or target type is not known statically, the OAD invariant cannot be guaranteed. A boundary check should be inserted in this case.

The type rules for type casts and boundary checks are present on Figure 8. It is important to notice, that for $\mathsf{JO}_?^+$ we use a different typing relations, namely, $\vdash^{\mathcal{C}}$ and $\vdash^{\mathcal{C}}_{\mathcal{B}}$. Generally, these two relations are similar to the original $\vdash$ for $\mathsf{JO}_?$. 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 rules $\lesssim$ is 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-CHECK) is targeted to ensure the OAD invariant. We postpone the detailed definition of the relations $\vdash^{\mathcal{C}}$ and $\vdash^{\mathcal{C}}_{\mathcal{B}}$ until Section 5.5.

5.5 Type-guided program translation

We adopt the idea of Siek and Taha [31] to define a type-directed *type cast* and *boundary check* insertion relation on expressions and methods (Figures 8, the relations $\overset{\mathcal{C}}{\leadsto}$ and $\overset{\mathcal{B}}{\leadsto}$, respectively). We distinguish insertions of type casts and boundary checks as two different procedures. First, the cast are inserted into a program. Boundary check insertion translation works subsequently on the program with inserted casts. This is an example of a *staged* translation, guided by the gradual type system: *each next step of the translation eliminates an aspect of uncertainty, caused by non-complete type annotations.*

Type cast insertion $\overset{\mathcal{C}}{\leadsto}$ is a first stage of the complete gradually-typed program translation. Figure 8 provides the definition of selected rules for the relation $E \vdash e_1 \overset{\mathcal{C}}{\leadsto} e_2 : t$. The relation subsumes the gradual type system and also specifies how to produce the translation. The translation can be performed only for well-typed expressions and methods.

Each inserted cast creates a fresh variable and increases the depth of the processed `let`-expression whenever the consistent-subtyping relation is mentioned in the premise of a typing rule. The cast insertion relation for expressions is written $E \vdash e_1 \overset{\mathcal{C}}{\leadsto} e_2 : t$ 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.

Type cast insertions are type-guided: we do not insert a cast or a boundary check if we get positive answers from the predicate $\lhd$. This is why we describe the helper function $\mathcal{C}$ that optionally inserts type-casts. We use the non-recursive *local decomposition* of an expression e via the context G.

The following lemma holds for the relation $\overset{\mathcal{C}}{\leadsto}$ with respect to the $\vdash^{\mathcal{C}}$-typing.

Lemma 5.8 ($\overset{\mathcal{C}}{\leadsto}$ is type-sound for expressions)**.** *If* $E \vdash e \overset{\mathcal{C}}{\leadsto} e' : s$ *then* $E \vdash^{\mathcal{C}} e' : s$.

Boundary check insertion $\overset{\mathcal{B}}{\leadsto}$ is a second stage of the whole translation. The only one type statement that can be affected by $\overset{\mathcal{B}}{\leadsto}$ 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 if not enough type informations about *primary* owners is provided. For the rest of statements, expressions and methods, $\overset{\mathcal{B}}{\leadsto}$ is just the identity. The translation $\overset{\mathcal{B}}{\leadsto}$ works on top of the $\vdash^{\mathcal{C}}$-well-typed program.

Lemma 5.9 ($\overset{\mathcal{B}}{\leadsto}$ is $\vdash^{\mathcal{C}}_{\mathcal{B}}$-sound for expressions)**.** *If* $E \vdash e' \overset{\mathcal{B}}{\leadsto} e'' : s$ *then* $E \vdash^{\mathcal{C}}_{\mathcal{B}} e'' : s$.

The complete translation of the gradually-typed program is defined as follows:

Definition 5.10 ($\leadsto$)**.** $E \vdash e \leadsto e' : s$ iff $E \vdash e \overset{\mathcal{C}}{\leadsto} e'' : s$ and $E \vdash e'' \overset{\mathcal{B}}{\leadsto} e' : s$ for some $e'' \in \mathsf{JO}^{+}_{?}$.

Theorem 5.11 (Complete program translation is $\vdash^{\mathcal{C}}_{\mathcal{B}}$-sound.)**.** $E \vdash e : s$ *iff* $E \vdash e \leadsto e' : s$ *and then* $E \vdash^{\mathcal{C}}_{\mathcal{B}} e' : s$.

5.6 Semantics of dynamic type casts and boundary checks

To define the procedure of checking 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. We define a helper relation to compute the dynamic structure of a type $H;\, B \vdash t \ltimes t'$ (Figure 8).

The statement $\widehat{H} \vdash s \lhd t'$ in the premise of the rule (CAST-CHECK) might seem odd since the check uses not the pure subtyping but the "more defined than" relation on types. However, there is nothing wrong since all owners of the left operand s are *known* and to satisfy the relation all the actual owners of its "upper-left" mediator should match actual owners of the type t'. The semantics of type cast *only cares about known owners in* t'. The test $\iota \prec^* o$ can be performed at run-time by checking whether o is ι or some transitive owner of ι—this information is available in objects.

6. Type Safety

The type safety of $\mathsf{JO}_?$ is 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. The translation relation $E \vdash e_1 \leadsto e_2 : s$ can be extended for classes and programs. For instance, we denote $P_1; e_1 \leadsto P_2; e_2$ if a program $P_2; e_2$ is obtained from $P_1; e_1$ by the compositional type-directed translation.

Proposition 6.1 (Check insertion and gradual typing)**.**

$$\vdash P; e \;\; iff \;\; \exists P', e'.\, P; e \leadsto P'; e'$$

The soundness result with respect to the OAD invariant relies on three facts:

1. An initial configuration of any program does obey the OAD invariant;
2. The subject reduction theorem guarantees the type preservation for subsequent configurations;
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 single-threaded heap, so need to separate environments for heap objects and references in stack frames. We define a heap typing environment $\mathcal{E}$ as follows:

$$\mathcal{E} \;\in\; \textbf{HeapEnv} \;\; ::= \;\; \emptyset \mid \mathcal{E}, \iota : c\langle k_{i \in 1..n}\rangle \mid \mathcal{E}, \iota \prec^* k$$

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. The rules in Figure 8 describe the relation of well-formed triples ($\mathcal{E}, E; B \vdash \diamond$) as well as equivalences between possible owners: static and dynamic ($\mathcal{E}, E; B \vdash r = r'$). We define well-typed heaps to connect heap typing environments with actual run-time heaps ($\mathcal{E} \vdash H$). The last clause $\widehat{H} \Rightarrow \mathcal{E}$ in the premise of the rule (HEAP) is the key ingredient to define *correct* run-time heaps (Definition 6.2). It states that the environment $\mathcal{E}$ provides no more information than can be obtained from the flattened heap via the standard rules.

Definition 6.2 (Heap entailment)**.** $\mathcal{E} \Rightarrow \mathcal{E}'$ iff $\mathcal{E}, \emptyset \vdash \mathcal{H}$ for all statements $\mathcal{H} \in \mathcal{E}'$

Stack typings are defined as lists of typing environments:

$$\overline{E} \;\in\; \textbf{StackEnv} \;\; ::= \;\; \textbf{Nil} \mid E \bullet \overline{E}$$

Stack typing is *well-formed* if all its constituents are well-formed. The functions head and tail are defined for stack typings as standard ones for lists. We use the notation $E_0 = \mathsf{head}(\overline{E})$, $E_1 = \mathsf{head}(\mathsf{tail}(\overline{E}))$ etc. Finally, we define the typing relation for pairs $\langle e, K\rangle$ where $e \in \textbf{Expr}$ and $K \in \textbf{Continuation}$ ($\mathcal{E}; \overline{E}; B \vDash \langle e, K\rangle$).

Lemma 6.3 (Initial state typing)**.** *$\mathcal{E}, E; B \vdash^{\mathcal{C}}_{\mathcal{B}} e : t$ iff*

$$\mathcal{E}, (E \bullet \mathbf{Nil}); B \vDash \langle H, B, e, \mathbf{mt} \rangle$$

for some initial heap H such that $\mathcal{E} \vdash H$.

Definition 6.4 (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 6.5 (Stack typing evolution)**.** We say that a stack typing $\overline{E}$ transforms to a stack typing $\overline{E}'$ (written $\overline{E} \rightsquigarrow \overline{E}'$) if one of the following holds:

- $\overline{E}' = E' \bullet \overline{E}$ for some E'
- $\overline{E}' = \mathsf{tail}(\overline{E})$
- $\overline{E}' = (E_0, x : t) \bullet \mathsf{tail}(\overline{E})$ for some t and $x \notin \mathrm{dom}(E_0)$

We use $\rightsquigarrow^*$ for the reflexive-transitive closure of $\rightsquigarrow$.

Theorem 6.6 states the subject reduction invariant.

Theorem 6.6 (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 6.7 is a key crucial for the type safety. It ensures that for all well-formed states, if it is possible to make a next step in the operational semantics, then the OAD invariant is preserved for the heap component of the resulting state.

Theorem 6.7 (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 $\mathsf{OAD}(H')$.*

We define the predicate NPE for *null-pointer error* on states as follows:

Definition 6.8 (Null-pointer error states)**.** The state $\mathcal{S} = \langle H, B, e, k \rangle$ is *stuck* because of *null-pointer dereferencing* ($\mathsf{NPE}(\mathcal{S})$) iff $e = D[y]$ for some y and $B(y) = \mathtt{null}$, where D is defined below:

$$D \quad ::= \quad \mathtt{let}\ x = [\,].m(y')\ \mathtt{in}\ e \ \mid\ \mathtt{let}\ x = ([\,].f = y')\ \mathtt{in}\ e \ \mid \\ \mathtt{let}\ x = [\,].f\ \mathtt{in}\ e$$

The NPE-states are *terminal* for execution traces in the provided semantics of $\mathsf{JO}_?/\mathsf{JO}^+_?$, since there is no transition rules for them. We avoid addressing `NonNull`-annotations and corresponding static safety results in this work.

Definition 6.9 (Initial state)**.** Assume $P; e$ to be a program in $\mathsf{JO}^+_?$, $H = \{\mathtt{world} \mapsto \bullet\}$, $B = \{\mathtt{this} \mapsto \mathtt{world}\}$ is an *initial binding environment*. Then the *initial configuration* of $P; e$ is $\mathbf{init}(e) = \langle H, B, e, \mathbf{mt} \rangle$.

Following [14], 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. One can see that taking $\mathcal{E} = \{\mathtt{world} : \mathtt{World}\}$ and $\overline{E} = \{\mathtt{this} : \mathtt{World}\} \bullet \mathbf{Nil}$, we obtain $\emptyset \vdash^{\mathcal{C}}_{\mathcal{B}} P; e \Rightarrow \mathcal{E}, \overline{E} \vDash \mathbf{init}(e)$ by Lemma 6.3. Theorem 6.10 ends our tool-chain of safety statements.

Theorem 6.10 (Static type safety of $\mathsf{JO}_?$)**.** *If $P; e \rightsquigarrow P'; e'$ 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 $b = z \leftarrow y$ for some H, B, t, y, z and K (OAD violation attempt).

Combined Theorems 6.7 and 6.10 state that the provided gradual type system ensures that (a) during a program execution no ownership invariant will be violated, and (b) *fully-annotated well-typed* programs will be executed until the result value or a *null-pointer error* state with no ownership invariant violation.

7. Related Work

The idea of combining static and dynamic type checking is close to the work of Flanagan on *hybrid types* [20]. 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. According to the classification proposed by Greenberg et al. [22], systems with hybrid and gradual types are related to the class *manifest* systems, i.e., those which contain additional constraints on data as part of types and enable a type checker to reason about them. Their counterparts are *latent* systems, in which contracts are purely dynamic checks [19].

Gordon and Noble in the work on dynamic ownership introduce ConstraintedJava, a scripting language that provides dynamic ownership checking [21]. The authors suggest a dynamic ownership structure consisting of an owner pointer in every object. Operations are provided to make use of and change these owner pointers. The semantics of the language relies on a message-passing protocol with a specific kind of monitoring.

Existential ownership types [24] are a mechanism that enables parameterisation of types, as well as owners, and enables variant subtyping of owners based on existential quantification [10]. This approach allows *owner-polymorphic methods* to be elegantly implemented and it distinguish objects with different and equal *unknown* owners. Existential quantification also helps to implement effective run-time downcasts in the presence of ownership types: subtype's inferred owners are treated as *existential* ones [35]. The key difference between these approaches and our approach is that existential ownership expresses *don't know* whereas gradual types express *don't care* concerning the unknown owners.

Algorithms for ownership inference are a domain of large interest nowadays. They 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 [34]. The work provides a graph-theoretical background for the runtime inference. The author formulates the system of equations to assign annotations to particular object allocation sites, based on an object graphs' evolution history. However, no proof of correctness of these equations is provided as well as it as conditions for the uniqueness of the solution.

Milanova and Liu present a static analyses to infer ownership and universe annotations according to two different ownership protocols: the owner-as-dominator and the owner-as-modifier [26]. Both analyses are based on the context-insensitive points-to analysis, therefore they do not distinguish between different allocation and call sites. However, thanks to some Java-related heuristics, the presented analyses handle some idiomatic cases, and better precision is obtained. The proof of correctness of the build dominance abstraction is present, although it does not rely on the abstract interpretation-like nature of the points-to analysis. More general points-to analysis-based algorithm to infer ownership and uniqueness is presented by Ma and Foster [25]. The algorithm combines constraint-based intraprocedural and interprocedural analyses. The collected information about encapsulation properties is not mapped to a type system. Points-to analysis-based approach for the general inference of type qualifiers in Java is also described by Greenfieldboyce and Foster [23]. The authors provide a formal framework JQual for type qualifier inference in a simple object-oriented language and use it for two particular applications. A user of the framework must, however, formulate the property of interest in terms of type qualifiers, not the program behaviour.

Moelius and Souter [27] employ a variation of an escape analysis [6] to infer ownership annotations. The presented algorithm allows borrowing references to be returned from methods and assigned to object fields. No assumptions on ownership parameterisation is made, consequently the algorithm results in a large number of parameters. Dietl et al. present a static analysis to infer Universe types [15] according to the set of generated constraints [17]. Constraints of the Universe type system are encoded as a boolean satisfiability problem. The described constraint-based analysis is close to traditional control-flow analyses via abstract interpretation. No formal proof of the correspondence of the obtained result to the original type system is provided.

Aldrich et al. present AliasJava, a capability-based alias annotation system for Java that makes alias patterns explicit in the source code [4]. The provided system of annotations allows both notions of uniqueness and ownership-style encapsulation to be captured. The programmer need only provide a small amount of annotations to indicate the intent to encapsulate some parts of the program and the rest of alias annotations will be inferred.

8. Conclusion

In this work we applied the notion of *gradual types* to ownership types systems and the ownership invariant in a Java-like language. The resulting formalism is an extension of the language JOE_1 by Clarke and Drossopoulou. We introduced the notion of *gradual ownership types* and corresponding *consistent-subtyping* relation to reason about ownership structures statically in the presence of statically unknown owners. For fully annotated programs the developed formalism provides static guarantee of the ownership invariant, whereas for partially annotated programs the dynamic checks necessary to ensure the invariant are inserted by the compiler.

In our work we mainly addressed extending a fixed host language (e.g., Java) with domain-specific annotations to ensure ownership and gradual migration from "raw" code to annotated code. Modern systems with aliasing-aware annotations usually require significant changes in the structure of the host language and its semantics [3, 12]. It is still a matter of discussion, how gradual typing could be useful for such specialized formalisms. The *Blame calculus* enables a programmer to reason about the causes of dynamic constraints violation via the mechanism of *label passing* and *variant blames* [2]. It would be interesting to explore the application of the formalism to object-oriented aliasing invariants.

Acknowledgements

We wish to acknowledge the valuable and very detailed comments of Sophia Drossopoulou on early drafts of the paper. We are also grateful to José Proença for proof-reading a draft version. Finally, we are grateful to the anonymous reviewers of IWACO '11 for their constructive feedback.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[2] A. Ahmed, R. B. Findler, J. G. Siek, and P. Wadler. Blame for all. In *POPL '11*, pages 201–214.

[3] J. Aldrich and C. Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *ECOOP 2004*, pages 1–25.

[4] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *OOPSLA '02*, pages 311–330.

[5] C. Anderson and S. Drossopoulou. BabyJ: From Object Based to Class Based Programming via Types. *ENTCS*, 82(8):53 – 81, 2003.

[6] B. Blanchet. Escape analysis: correctness proof, implementation and experimental results. In *POPL '98*, pages 25–37.

[7] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *OOPSLA '01*, pages 56–69.

[8] 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.

[9] N. Cameron. *Existential Types for Variance – Java Wildcards and Ownership Types*. PhD thesis, Imperial College London, London, UK, 2009.

[10] N. Cameron and S. Drossopoulou. Existential quantification for variant ownership. In *ESOP '09*, pages 128–142.

[11] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA '02*, pages 292–310.

[12] D. Clarke and T. Wrigstad. External uniqueness is unique enough. In *ECOOP 2003*, pages 176–200.

[13] D. G. Clarke. *Object ownership and containment*. PhD thesis, University of New South Wales, New South Wales, Australia, 2003.

[14] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA '98*, pages 48–64.

[15] D. Cunningham, W. Dietl, S. Drossopoulou, A. Francalanza, P. Müller, and A. J. Summers. In *FMCO 2007*, chapter Universe Types for Topology and Encapsulation, pages 72–112.

[16] O. Danvy. Defunctionalized interpreters for programming languages. In *ICFP '08*, pages 131–142.

[17] W. Dietl, M. D. Ernst, and P. Müller. Tunable Universe type inference. Technical Report 659, Department of Computer Science, ETH Zurich, Dec. 2009.

[18] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex, 1st edition*. The MIT Press, August 2009.

[19] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ICFP '02*, pages 48–59.

[20] C. Flanagan. Hybrid type checking. In *POPL '06*, pages 245–256.

[21] D. Gordon and J. Noble. Dynamic ownership in a dynamic language. In *DLS '07*, pages 41–52.

[22] M. Greenberg, B. C. Pierce, and S. Weirich. Contracts made manifest. In *POPL '10*, pages 353–364.

[23] D. Greenfieldboyce and J. S. Foster. Type qualifier inference for Java. In *OOPSLA '07*, pages 321–336.

[24] Y. Lu and J. Potter. On ownership and accessibility. In *ECOOP 2006*, pages 99–123.

[25] K.-K. Ma and J. S. Foster. Inferring aliasing and encapsulation properties for Java. In *OOPSLA '07*, pages 423–440.

[26] A. Milanova and Y. Liu. Practical static ownership inference. Technical report, Rensselaer Polytechnic Institute, Troy NY 12110, USA, 2010.

[27] S. E. Moelius III and A. L. Souter. An object ownership inference algorithm and its applications. In *MASPLAS '04*.

[28] P. Müller. In *VSTTE '05*, chapter Reasoning about Object Structures Using Ownership, pages 93–104.

[29] J. Östlund and T. Wrigstad. Welterweight Java. In *TOOLS '10*, pages 97–116.

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

[31] J. Siek and W. Taha. Gradual typing for objects. In *ECOOP 2007*, pages 2–27, .

[32] J. Siek and W. Taha. Gradual typing for functional languages. In *SCHEME '06*, pages 81–92, .

[33] J. Vitek and B. Bokowski. Confined types. In *OOPSLA '99*, pages 82–96.

[34] A. Wren. Inferring ownership. Master's thesis, Imperial College London, London, UK, June 2003.

[35] T. Wrigstad and D. Clarke. Existential owners for ownership types. *Journal of Object Technology*, 6(4):141–159, 2007.

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

$$\text{(OWN-WORLD)}\ \frac{E;B \vdash \diamond}{E;B \vdash \texttt{world}} \qquad \text{(OWN-VARLOC)}\ \frac{E;B \vdash w : t}{E;B \vdash w} \qquad \text{(OWN-UNKNOWN)}\ \frac{E;B \vdash \diamond}{E;B \vdash ?}$$

$$\text{(OWN-DEPENDENT)}\ \frac{i \in 1..\mathsf{arity}(c)}{E;B \vdash x^{c.i}} \qquad \text{(OWN-VAR)}\ \frac{E;B \vdash \diamond \quad \alpha \prec^* p \in E}{E;B \vdash \alpha}$$

$$\boxed{E;B \vdash p \prec^* p'}$$

$$\text{(IN-ENV)}\ \frac{E;B \vdash \diamond \quad \alpha \prec^* r \in E}{E;B \vdash \alpha \prec^* r} \qquad \text{(IN-REFL)}\ \frac{E;B \vdash r}{E;B \vdash r \prec^* r}$$

$$\boxed{E;B \vdash p \prec^* p'}$$

$$\text{(IN-WORLD)}\ \frac{E;B \vdash r}{E;B \vdash r \prec^* \texttt{world}} \qquad \text{(IN-TRANS)}\ \frac{E;B \vdash r \prec^* r' \quad E;B \vdash r' \prec^* r''}{E;B \vdash r \prec^* r''} \qquad \text{(IN-THIS)}\ \frac{E;B \vdash w : t \quad r{=}\mathsf{owner}(t)}{E;B \vdash w \prec^* r}$$

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

$$\text{(SUB-DEP1)}\ \frac{E;B \vdash x^{c.i}}{E;B \vdash p \preceq x^{c.i}} \qquad \text{(SUB-DEP2)}\ \frac{E;B \vdash x^{c.i}}{E;B \vdash x^{c.i} \preceq p} \qquad \text{(SUB-LEFT)}\ \frac{E;B \vdash p}{E;B \vdash p \preceq ?}$$

$$\text{(SUB-RIGHT)}\ \frac{E;B \vdash p}{E;B \vdash ? \preceq p} \qquad \text{(SUB-INCL)}\ \frac{E;B \vdash r \prec^* r'}{E;B \vdash r \preceq r'}$$

Figure 4. Well-formed owners and owner ordering

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

$$\text{(CON-REFL)}\ \frac{E;B \vdash p}{E;B \vdash p \sim p} \qquad \text{(CON-RIGHT)}\ \frac{E;B \vdash p}{E;B \vdash ? \sim p} \qquad \text{(CON-LEFT)}\ \frac{E;B \vdash p}{E;B \vdash p \sim ?}$$

$$\text{(CON-DEPENDENT1)}\ \frac{E;B \vdash p \quad E;B \vdash x^{c.i}}{E;B \vdash p \sim x^{c.i}} \qquad \text{(CON-DEPENDENT2)}\ \frac{E;B \vdash p \quad E;B \vdash x^{c.i}}{E;B \vdash x^{c.i} \sim p}$$

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

$$\text{(CON-TYPE)}\ \frac{E;B \vdash c\langle p_{i\,\in\,1..n}\rangle \quad E;B \vdash c\langle q_{i\,\in\,1..n}\rangle \quad p_i \sim q_i,\ \forall\, i{\in}1..n}{E;B \vdash c\langle p_{i\,\in\,1..n}\rangle \sim c\langle q_{i\,\in\,1..n}\rangle}$$

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

$$\text{(SUB-REFL)}\ \frac{E;B \vdash t}{E;B \vdash t \leq t} \qquad \text{(SUB-TRANS)}\ \frac{E;B \vdash t \leq t' \quad E;B \vdash t' \leq t''}{E;B \vdash t \leq t''}$$

$$\text{(SUB-CLASS)}\ \frac{E;B \vdash c\langle\sigma\rangle \quad \texttt{class}\ c\langle\alpha_{i\in1..n}\rangle\ \texttt{extends}\ c'\langle r_{i\in1..n'}\rangle\{\ldots\}}{E;B \vdash c\langle\sigma\rangle \leq c'\langle\sigma(r_i)_{i\in1..n}\rangle}$$

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

$$\text{(GRAD-SUB)}\ \frac{E;B \vdash c\langle\sigma\rangle \leq c'\langle\sigma'\rangle \quad E;B \vdash c'\langle\sigma'\rangle \sim c'\langle\sigma''\rangle}{E;B \vdash c\langle\sigma\rangle \lesssim c'\langle\sigma''\rangle}$$

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

$$\text{(G-TYPE)}\ \frac{\mathsf{arity}(c){=}n \quad E;B \vdash p_1 \preceq p_i \quad \forall i \in 1..n}{E;B \vdash c\langle p_{i\,\in\,1..n}\rangle}$$

Figure 5. Type consistency and subtyping

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

$$\text{(T-NEW)}\ \frac{E;B \vdash c\langle r_{i\in1..n}\rangle}{E;B \vdash \texttt{new}\ c\langle r_{i\in1..n}\rangle : c\langle r_{i\in1..n}\rangle} \qquad \text{(T-LKP)}\ \frac{E;B \vdash z : c\langle\sigma\rangle \quad \mathcal{F}_c(f){=}t}{E;B \vdash z.f : \sigma_z(t)}$$

$$\text{(T-LET)}\ \frac{E;B \vdash b : t \quad E,x : \mathsf{fill}(x,t);B \vdash e : s}{E;B \vdash \texttt{let}\ x{=}b\ \texttt{in}\ e : s}$$

$$\text{(T-UPD)}\ \frac{E;B \vdash z : c\langle\sigma\rangle \quad \mathcal{F}_c(f){=}t \quad E;B \vdash y : s \quad E;B \vdash s \lesssim \sigma_z(t)}{E;B \vdash z.f{=}y : \sigma_z(t)}$$

$$\text{(T-CALL)}\ \frac{E;B \vdash y : s \quad \mathcal{MT}_c(m){=}(y',t{\to}t') \quad E;B \vdash z : c\langle\sigma\rangle \quad E;B \vdash s \lesssim \sigma_z(t) \quad \sigma'{\equiv}\sigma{\uplus}\{y'{\mapsto}y\}}{E;B \vdash z.m(y) : \sigma'_z(t')}$$

$$\boxed{E \vdash t'\ m(t\ y)\ \{e\}} \qquad \boxed{E \vdash c} \qquad \boxed{\vdash P;\, e}$$

$$\text{(VAL-}w\text{)}\ \frac{E;B \vdash \diamond \quad w : s \in E}{E;B \vdash w : s} \qquad \text{(VAL-NULL)}\ \frac{E;B \vdash t}{E;B \vdash \texttt{null} : t}$$

$$\text{(METHOD)}\ \frac{E,y : \mathsf{fill}(y,t) \vdash e : s \quad E \vdash s \lesssim t'}{E \vdash t'\ m(t\ y)\ \{e\}} \qquad \text{(CLASS-OBJECT)}\ \frac{}{\vdash \texttt{class Object}\langle\alpha_1\rangle\ \{\ \}}$$

$$\text{(CLASS)}\ \frac{\begin{array}{c} E{\equiv}\alpha_1 \prec^* \texttt{world},(\alpha_1 \prec^* \alpha_i)_{i\,\in\,2..n},\texttt{this} : c\langle\alpha_{i\in1..n}\rangle \\ E \vdash c'\langle\sigma\rangle \qquad \mathsf{owner}(c\langle\alpha_{i\in1..n}\rangle){=}\mathsf{owner}(c'\langle\sigma\rangle) \\ \{f_{i\in1..m}\}\cup(\mathcal{F}_{c'}){=}\emptyset \qquad E \vdash t_{j\in1..m} \qquad E \vdash meth_{k\in1..p} \\ \forall\, m \in \begin{array}{l}\mathsf{names}(meth_{k\in1..p}) \\ \cup\ \mathrm{dom}(\mathcal{MT}_{c'})\end{array} \left\{\begin{array}{l} \mathcal{MT}_c(m){\equiv}t{\to}t' \\ \mathcal{MT}_{c'}(m){\equiv}t''{\to}t''' \\ t{\equiv}\sigma(t'') \quad t'{\equiv}\sigma(t''') \end{array}\right. \end{array}}{\vdash \texttt{class}\ c\langle\alpha_{i\in1..n}\rangle\ \texttt{extends}\ c'\langle\sigma\rangle\ \{t_j\ f_{j\in1..m}\ meth_{k\in1..p}\}}$$

$$\text{(PROGRAM)}\ \frac{\vdash class_j \quad \forall class_j \in P \quad E \vdash e : t}{E \vdash P;\, e}$$

Figure 6. Typing rules

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

$$\text{(E-LKP)}\ \frac{B(y){=}\iota \qquad H(\iota).f{=}v}{\langle H,B,\texttt{let}\ x{=}y.f\ \texttt{in}\ e,K\rangle{\Rightarrow}\langle H,B[x{\mapsto}v],e,K\rangle}$$

$$\text{(E-UPD)}\ \frac{B(y){=}\iota \quad B(y'){=}v \quad H(\iota){=}o \quad H'{=}H \uplus \iota{\mapsto}o[f{\mapsto}v]}{\langle H,B,\texttt{let}\ x{=}(y.f{=}y')\ \texttt{in}\ e,K\rangle{\Rightarrow}\langle H',B[x{\mapsto}v],e,K\rangle}$$

$$\text{(E-NEW)}\ \frac{\iota \text{ is fresh} \quad H'{\equiv}H \uplus \iota{\mapsto}\langle c\langle B(r_i)_{i\,\in\,1..n}\rangle,[f{\mapsto}\texttt{null}_{f\,\in\,\mathrm{dom}(\mathcal{F}_c)}]\rangle}{\langle H,B,\texttt{let}\ x{=}\texttt{new}\ c\langle r_{i\,\in\,1..n}\rangle\ \texttt{in}\ e,K\rangle{\Rightarrow}\langle H',B[x{\mapsto}\iota],e,K\rangle}$$

$$\text{(E-CALL)}\ \frac{\begin{array}{c} B(y){=}\iota \quad B(y'){=}v \quad H(\iota){=}\langle c\langle\sigma'\rangle,[f{\mapsto}v_{f\,\in\,\mathrm{dom}(\mathcal{F}_c)}]\rangle \\ \mathcal{M}_c(m){=}(x',e',\{\alpha_i{\mapsto}r_{i\,\in\,1..n}\}) \\ B'{=}\{\alpha_i{\mapsto}\sigma'(r_i)_{i\,\in\,1..n},\texttt{this}{\mapsto}\iota,x'{\mapsto}v\} \end{array}}{\langle H,B,\texttt{let}\ x{:}[\![t,\sigma]\!]{=}y.m(y')\ \texttt{in}\ e,K\rangle{\Rightarrow}\langle H,B',e',\mathbf{call}(x{:}[\![t,\sigma]\!],e,B,K)\rangle}$$

$$\text{(E-RETURN)}\ \langle H,B,y,\mathbf{call}(x{:}[\![t,\sigma]\!],e,B',K)\rangle{\Rightarrow}\langle H,B'[x{\mapsto}B(y)],e,K\rangle \qquad \text{(E-FINAL)}\ \langle H,B,y,\mathbf{mt}\rangle{\Rightarrow}\langle H,B,B(y),\mathbf{mt}\rangle$$

Figure 7. Small-step operational semantics of JO?

Local context

$G ::= [\,] \mid \texttt{let}\ x{=}z.m([\,])\ \texttt{in}\ e \mid \texttt{let}\ x{=}(z.f{=}[\,])\ \texttt{in}\ e$

Return context

$F ::= [\,] \mid \texttt{let}\ z{=}b\ \texttt{in}\ F$

Conditional cast insertion

$$\begin{array}{ll}\mathcal{C}_E\langle t_1,t_2\rangle \triangleq & \lambda e.\ \text{if } t_1 \lhd t_2 \text{ then } e\\ & \text{else } \texttt{let}\ y'{=}\langle t_2\rangle y\ \texttt{in}\ G[y']\\ & \text{where } y' \text{ is fresh, } e{=}G[y]\end{array}$$

Conditional check insertion

$\mathcal{B}_E\langle t\rangle \triangleq \lambda b@(z.f{=}y).\ \text{if } \textbf{specified}(t) \text{ then } b \text{ else } z.f{\leftarrow}y$

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

(C-UPD)

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

(C-METHOD)

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

(C-CALL)

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

(B-UPD)

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

$\boxed{H;\, B \vdash t \ltimes t'}$ $\boxed{\langle H,B,e,K\rangle \Rightarrow \langle H',B',e',K'\rangle}$

(CAST-CHECK)

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

(BOUNDARY-CHECK)

$$\frac{B(x){=}\iota \quad B(y){=}\iota' \\ H(\iota'){=}\langle c\langle o,\ldots\rangle,[\ldots]\rangle \quad \widehat{H};\emptyset \vdash \iota{\prec^*}o}{H;B \vdash \mathsf{boundary}(x,y)}$$

(TYPE-INSTANCE)

$$\frac{\forall\ i \in 1..n \quad q_i{=}\begin{cases} k & \text{if } \begin{cases} p_i{=}x^{c.j} \\ H(B(x)){=}\langle t,[\ldots]\rangle \\ k{=}\mathsf{owner}_j(t{\uparrow}c)\end{cases} \\ p_i & \text{if } p_i \text{ is } \textit{actual} \\ B(p_i) & \text{if } p_i \text{ is } \textit{concrete} \\ ? & \text{otherwise}\end{cases}}{H;B \vdash c\langle p_{i\in 1..n}\rangle \ltimes c\langle q_{i\in 1..n}\rangle}$$

(E-CAST1)

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

(E-BOUNDARY1)

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

(E-CAST2)

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

(E-BOUNDARY2)

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

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

(T-CAST)

$$\frac{E;B \vdash y : s \quad E;B \vdash t \\ E;B \vdash s \lesssim t}{E;B \vdash^{\mathcal{C}} \langle t\rangle y : t}$$

(T-UPD')

$$\frac{E;B \vdash z : c\langle\sigma\rangle \quad \mathcal{F}_c(f){=}t \\ E;B \vdash s \lhd \sigma_z(t) \quad E;B \vdash y : s}{E;B \vdash^{\mathcal{C}} z.f{=}y : \sigma_z(t)}$$

(T-CHECK)

$$\frac{E;B \vdash z : c\langle\sigma\rangle \quad \mathcal{F}_c(f){=}t \\ E;B \vdash y : s \quad E;B \vdash s \lhd \sigma_z(t)}{E;B \vdash^{\mathcal{C}}_{\mathcal{B}} z.f{\leftarrow}y : \sigma_z(t)}$$

(T-CALL')

$$\frac{E;B \vdash y : s \quad \mathcal{MT}_c(m){=}(y',t{\to}t') \\ E;B \vdash z : c\langle\sigma\rangle \quad E;B \vdash s \lhd \sigma_z(t) \\ \sigma'{\equiv}\sigma \uplus \{y' \mapsto y\}}{E;B \vdash^{\mathcal{C}} z.m(y) : \sigma'_z(t')}$$

(METHOD')

$$\frac{E,y : \mathsf{fill}(y,t);B \vdash e : s \\ E;B \vdash s \lhd t'}{E;B \vdash^{\mathcal{C}} t'\ m(t\ y)\ \{e\}}$$

(T-UPD")

$$\frac{E;B \vdash z : c\langle\sigma\rangle \quad \mathcal{F}_c(f){=}t \quad E;B \vdash y : s \\ E;B \vdash s \lhd \sigma_z(t) \quad \textbf{specified}(\sigma_z(t))}{E;B \vdash^{\mathcal{C}}_{\mathcal{B}} z.f{=}y : \sigma_z(t)}$$

$\boxed{\mathcal{E},E;B \vdash \diamond}$ $\boxed{\mathcal{E},E;B \vdash r = r'}$

(BINDING-OWNER)

$$\frac{\alpha\ \mathsf{R}\ r \in E \quad \mathcal{E},E;B \vdash k\ \mathsf{R}\ r \\ \alpha \notin \mathrm{dom}(B) \quad \mathsf{R} \in \{\prec^*,\succ^*\}}{\mathcal{E},E;B,\alpha{=}k \vdash \diamond}$$

(BINDING-VALUE)

$$\frac{\mathcal{E},E;B \vdash v : s \quad z : s' \in E \quad s'{=}c\langle\sigma\rangle \quad z \notin \mathrm{dom}(B) \\ \mathcal{E},E;B \vdash s \leq c\langle\sigma \uplus \{z^{c.j} \mapsto \mathsf{owner}_j(s{\uparrow}c)\}\rangle}{\mathcal{E},E;B,z{=}v \vdash \diamond}$$

(IN-BIND1)

$$\frac{\mathcal{E},E;B \vdash \diamond \\ \alpha{=}k \in B}{\mathcal{E},E;B \vdash \alpha{=}k}$$

(IN-BIND2)

$$\frac{\mathcal{E},E;B \vdash \diamond \\ z{=}\iota \in B}{\mathcal{E},E;B \vdash z{=}\iota}$$

(IN-BIND3)

$$\frac{\mathcal{E},E;B \vdash \diamond \quad \mathcal{E},E;B \vdash z{=}v \\ \mathcal{E},E;B \vdash z : c\langle\sigma\rangle \quad \mathcal{E},E;B \vdash v : s}{\mathcal{E},E;B \vdash z^{c.i}{=}\mathsf{owner}_i(s{\uparrow}c)}$$

$\boxed{\mathcal{E} \vdash \iota \mapsto o\ : s}$ $\boxed{\mathcal{E} \vdash H}$ $\boxed{\mathcal{E};\overline{E};B \vDash \langle e,K\rangle}$

(HEAP-OBJECT)

$$\frac{o{\equiv}\langle c\langle\sigma\rangle,[f \mapsto v_{f\ \in\ \mathrm{dom}(\mathcal{F}_c)}]\rangle \\ \mathcal{E};\emptyset \vdash c\langle\sigma\rangle \quad \mathcal{E};\emptyset \vdash \iota{\prec^*}\mathsf{owner}(c\langle\sigma\rangle) \\ \mathcal{E};\emptyset \vdash v_f : s \quad \mathcal{E} \vdash s \lhd \sigma_\iota(\mathcal{F}_c(f)) \quad \forall f \in \mathrm{dom}(\mathcal{F}_c)}{\mathcal{E} \vdash \iota \mapsto o : c\langle\sigma\rangle}$$

(HEAP)

$$\frac{\iota{:}t \in \mathcal{E} \quad \mathcal{E} \vdash \iota \mapsto o{:}t \\ \forall \iota \mapsto o \in H \quad \widehat{H}{\Rightarrow}\mathcal{E}}{\mathcal{E} \vdash H}$$

(TC-MT)

$$\frac{\mathcal{E},E;B \vdash^{\mathcal{C}}_{\mathcal{B}} e : s}{\mathcal{E};E{\bullet}\textbf{Nil};B \vDash \langle e,\textbf{mt}\rangle}$$

(TC-FAIL)

$$\frac{\mathcal{E};\overline{E};B \vDash \langle e,K\rangle}{\mathcal{E};\overline{E};B \vDash \langle e,\textbf{fail}(K)\rangle}$$

(TC-CALL)

$$\frac{\mathcal{E},E_0;B \vdash^{\mathcal{C}}_{\mathcal{B}} e : s \quad \mathcal{E},E_0;B \vdash s \lhd t \\ \forall r \in \mathrm{dom}(\sigma) \quad (\mathcal{E},E_0,B \vdash r{=}r') {\Leftrightarrow} (\mathcal{E},E_1,B' \vdash \sigma(r){=}r') \\ \mathcal{E},E_1,(x : \mathsf{fill}(x,\sigma(t))),\overline{E};B' \vDash \langle e',K\rangle}{\mathcal{E};E_0{\bullet}E_1{\bullet}\overline{E};B \vDash \langle e,\textbf{call}(x : [\![t,\sigma]\!],e',B',K)\rangle}$$

$\boxed{\mathcal{E},\overline{E} \vDash \langle H,B,e,K\rangle}$

(T-STATE)

$$\frac{\mathcal{E} \vdash H \quad \mathcal{E},\overline{E};B \vDash \langle e,K\rangle}{\mathcal{E},\overline{E} \vDash \langle H,B,e,K\rangle}$$

Figure 8. Casts and boundary checks: auxiliary definitions, the translation and the extended small-step operational semantics. Selected typing rules of $\vdash^{\mathcal{C}}$ and $\vdash^{\mathcal{C}}_{\mathcal{B}}$, well-formed bindings, heaps and continuations.