Safe Runtime Downcasts With Ownership Types

Chandrasekhar Boyapati, Robert Lee, and Martin Rinard

Laboratory for Computer Science
Massachusetts Institute of Technology
200 Technology Square, Cambridge, MA 02139
{chandra,rhlee,rinard}@lcs.mit.edu

Abstract. The possibility of aliasing between objects constitutes one of the primary challenges in understanding and reasoning about correctness of object-oriented programs. Ownership types provide a principled way of specifying statically enforcable restrictions on object aliasing. Ownership types have been used to aid program understanding and evolution, verify absence of data races and deadlocks in multithreaded programs, and verify absence of memory errors in programs with explicit deallocation.

This paper describes an efficient technique for supporting safe runtime downcasts with ownership types. This technique uses the type passing approach, but avoids the associated significant space overhead by storing runtime ownership information only for objects that are potentially involved in downcasts. Moreover, this technique does not use any interprocedural analysis, so it preserves the separate compilation model of Java. We implemented our technique in the context of Safe Concurrent Java, which is an extension to Java that uses ownership types to guarantee the absence of data races and deadlocks in well-typed programs. Our approach is JVM-compatible: our implementation translates programs to bytecodes that can be run on regular JVMs.

1 Introduction

The possibility of aliasing between objects constitutes one of the primary challenges in understanding and reasoning about correctness of object-oriented programs [14]. Unexpected aliasing can lead to broken invariants, mistaken assumptions, security holes, and surprising side effects, all of which may lead to defective software. Ownership types provide a principled way of specifying statically enforceable restrictions on object aliasing.

Ownership types were first introduced in Flexible Alias Protection [9], and formalized in [8]. Parameterized Race Free Java (PRFJ) [5] extends ownership types to support inheritance and dynamic aliases (which allow, e.g., the description of iterators) and uses them to statically ensure the absence of data races in Java programs. PRFJ also uses effects clauses and combines ownership types with unique pointers [16], and read-only fields and objects, which allows many important idioms to be expressed. Safe Concurrent Java (SCJ) [4] extends PRFJ to prevent both data races and deadlocks in Java programs. AliasJava [2] uses ownership types to aid software evolution. Cyclone [13] uses a similar type system to guarantee absence of memory errors in programs with explicit deallocation.

In ownership type systems, programmers can parameterize classes and methods by owners. This enables the writing of generic code that can be used in many different contexts. The parameterization is somewhat similar to the proposals for parametric types for Java [17, 6, 1, 19]. Ownership type systems are primarily static type systems. The type checker uses the ownership type annotations to statically ensure the absence of certain classes of errors (e.g., data races in PRFJ, memory errors in Cyclone), but it is usually unnecessary to preserve the ownership information at runtime. However, languages like Java [12] are not purely statically typed languages. Java allows downcasts that are checked at runtime. To support safe runtime downcasts, the system must preserve some ownership information at runtime when ownership types are used in the context of a language like Java.

There are primarily three techniques used to implement parametric polymorphism in a language like Java. The *type erasure* approach [6, 7] is based on the idea of deleting type parameters (so Stack⟨T⟩ erases to Stack). But this approach will not preserve ownership information at runtime, so it is unsuitable for supporting safe runtime downcasts with ownership types. In the *code duplication* approach [1], polymorphism is supported by creating specialized classes/methods, each supporting a different instantiation of a parametric class/method. But since the parameters in ownership types are usually objects, this approach will lead to an unacceptably large number of classes/methods. In the *type passing* approach [17, 19, 18], information on type parameters is explicitly stored in objects and passed to code requiring them. But if the system stores the owners of every object at runtime, this approach has the potential drawback of adding a per-object space overhead. Java objects are typically very small, so adding even a single field to every object may increase the size of most objects by a significant fraction.

This paper describes an efficient technique for supporting safe runtime downcasts with ownership types. This technique uses the type passing approach, but avoids the associated significant space overhead by storing runtime ownership information only for objects that are potentially involved in downcasts. Moreover, this technique does not use any inter-procedural analysis, so it preserves the separate compilation model of Java. We implemented our technique in Safe Concurrent Java [4, 5], which is an extension to Java that uses ownership types to guarantee the absence of data races and deadlocks in well-typed programs. Our approach is JVM-compatible: our implementation translates programs to bytecodes that can be run on regular JVMs [15].

The rest of this paper is organized as follows. Section 2 gives an overview of ownership types in Safe Concurrent Java (SCJ). Section 3 describes how we support safe runtime downcasts in SCJ. Section 4 presents related work, and Section 5 concludes.

2 Safe Concurrent Java

Safe concurrent Java (SCJ) [4, 5] is an extension to Java that guarantees the absence of data races and deadlocks in well-typed programs. The basic idea behind our system is as follows. When programmers write multithreaded programs, they already have a locking discipline in mind. Our system allows programmers to specify this locking discipline in their programs. The resulting specifications take the form of type declarations.

To prevent data races in SCJ, programmers associate every object with a *protection mechanism* that ensures that accesses to the object never create data races. The protection mechanism of an object can specify either the mutual exclusion lock that protects the object from unsynchronized concurrent accesses, or that threads can safely access the object without synchronization because either 1) the object is immutable, 2) the object is accessible to a single thread, or 3) the variable contains the unique pointer to the object. Unique pointers are useful to support object migration between threads. The type checker statically verifies that a program uses objects only in accordance with their declared protection mechanisms.

To prevent deadlocks, programmers partition all the locks into a fixed number of lock levels and specify a partial order among the lock levels. The type checker statically verifies that whenever a thread holds more than one lock, the thread acquires the locks in the descending order. SCJ also allows programmers to use recursive tree-based data structures to further order the locks that belong to the same lock level. For example, programmers can specify that nodes in a tree must be locked in the *tree-order*. SCJ allows mutations to the data structure that change the partial order at runtime. The type checker uses an intra-procedural intra-loop flow-sensitive analysis to statically verify that the mutations do not introduce cycles in the partial order, and that the changing of the partial order does not lead to deadlocks.

2.1 Ownership Types in a Subset of Safe Concurrent Java

This section presents Mini Safe Concurrent Java (MSCJ), which is a subset of SCJ that prevents data races in well-typed programs. To simplify the presentation of key ideas behind our approach, the rest of the discussion in this paper will be in the context of MSCJ. Our implementation, however, works for the whole of SCJ and handles all the features of the Java language. The key to the MSCJ type system is the concept of object ownership. Every object in MSCJ has an owner. An object can be owned by another object, by itself, or by a special per-thread owner called `thisThread`. Objects owned by `thisThread`, either directly or transitively, are local to the corresponding thread and cannot be accessed by any other thread. Figure 1 presents an example ownership relation. We draw an arrow from object x to object y in the figure if object x owns object y. Our type system statically verifies that a program respects the ownership properties shown in Figure 2.

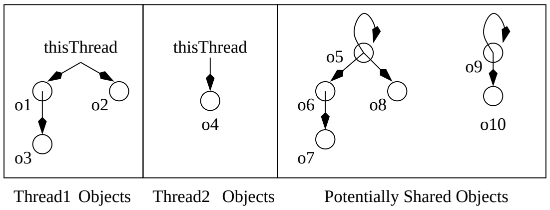

Fig. 1. An Ownership Relation

1. The owner of an object does not change over time.
2. The ownership relation forms a forest of rooted trees, where the roots can have self loops.
3. The necessary and sufficient condition for a thread to access to an object is that the thread must hold the lock on the root of the ownership tree that the object belongs to.
4. Every thread implicitly holds the lock on the corresponding thisThread owner. A thread can therefore access any object owned by its corresponding thisThread owner without any synchronization.

Fig. 2. Ownership Properties

Figure 3 shows the grammar for MSCJ. Figure 4 shows a TStack program in MSCJ. For simplicity, all the examples in this paper use an extended language that is syntactically closer to Java. A TStack is a stack of T objects. A TStack is implemented using a linked list. A class definition in MSCJ is parameterized by a list of owners. This parameterization helps programmers write generic code to implement a class, then create different objects of the class that have different protection mechanisms. In Figure 4, the TStack class is parameterized by thisOwner and TOwner. thisOwner owns the this TStack object and TOwner owns the T objects contained in the TStack. In general, the first formal parameter of a class always owns the this object. In case of s1, the owner thisThread is used for

$$\begin{array}{rcl}
P & ::= & \mathit{defn}^{*}\ e \\
\mathit{defn} & ::= & \textsf{class}\ \mathit{cn}\langle \mathit{owner}\ f^{*}\rangle\ \textsf{extends}\ c\ \{\mathit{field}^{*}\ \mathit{meth}^{*}\} \\
c & ::= & \mathit{cn}\langle \mathit{owner}{+}\rangle \mid \textsf{Object}\langle \mathit{owner}{+}\rangle \\
\mathit{owner} & ::= & f \mid \textsf{self} \mid \textsf{thisThread} \mid e_{\mathrm{final}} \\
\mathit{meth} & ::= & t\ \mathit{mn}(\mathit{arg}^{*})\ \textsf{accesses}\ (e_{\mathrm{final}}{}^{*})\ \{e\} \\
\mathit{field} & ::= & [\textsf{final}]_{\mathrm{opt}}\ t\ \mathit{fd} = e \\
\mathit{arg} & ::= & [\textsf{final}]_{\mathrm{opt}}\ t\ x \\
t & ::= & c \mid \textsf{int} \\
 & & \\
e & ::= & \textsf{new}\ c \mid x \mid x = e \mid e.\mathit{fd} \mid e.\mathit{fd} = e \mid e.\mathit{mn}(e^{*}) \mid e;e \mid \textsf{let}\ (\mathit{arg}{=}e)\ \textsf{in}\ \{e\} \mid \\
 & & \textsf{synchronized}\ (e)\ \textsf{in}\ \{e\} \mid \textsf{fork}\ (x^{*})\ \{e\} \\
e_{\mathrm{final}} & ::= & e \\
 & & \\
\mathit{cn} & \in & \text{class names}, \mathit{fd} \in \text{field names}, \mathit{mn} \in \text{method names}, x \in \text{variable names}, f \in \text{owner names}
\end{array}$$

Fig. 3. MSCJ Grammar

```
// thisOwner owns the TStack object, TOwner owns the T objects in the stack.
class TStack<thisOwner, TOwner> {
  TNode<this, TOwner> head = null;
  TStack() {}
  void push(T<TOwner> value) accesses (this) {
    TNode<this, TOwner> newNode = new TNode<this, TOwner>(value, head); head = newNode;
  }
  T<TOwner> pop() accesses (this) {
    T<TOwner> value = head.value(); head = head.next(); return value;
  }
}
class TNode<thisOwner, TOwner> {
  T<TOwner> value; TNode<thisOwner, TOwner> next;
  TNode(T<TOwner> v, TNode<thisOwner, TOwner> n) accesses (this) {
    this.value = v; this.next  = n;
  }
  T<TOwner> value() accesses (this) { return value; }
  TNode<thisOwner, TOwner> next() accesses (this) { return next; }
}
class T<thisOwner> { int x=0; }

TStack<thisThread, thisThread> s1 = new TStack<thisThread, thisThread>;
TStack<thisThread, self>       s2 = new TStack<thisThread, self>;
```

Fig. 4. Stack of T Objects in MSCJ

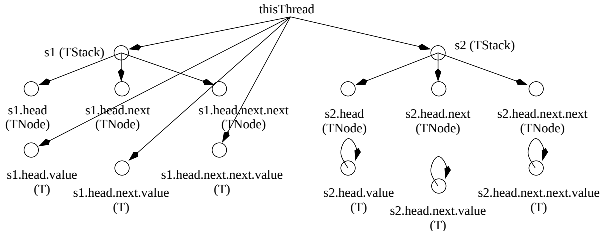

Fig. 5. Ownership Relation for TStacks s1 and s2

both the parameters to instantiate the TStack class. This means that the main thread owns TStack s1 as well as all the T objects contained in the TStack. In case of s2, the main thread owns the TStack but the T objects contained in the TStack own themselves. The ownership relation for the TStack objects s1 and s2 is depicted in Figure 5 (assuming the stacks contain three elements each). In MSCJ, a method can contain an accesses clause that specifies the objects the method accesses that must be protected by externally acquired locks. Callers are required to hold the locks on the root owners of the objects specified in the accesses clause before they invoke a method. In the example, the value and next methods in the TNode class assume that the callers hold the lock on the root owner of the this TNode object. Without the accesses clause, the value and next methods would not have been well-typed.

2.2 Static Type Checking

This section describes some of the important rules for static type checking of ownership types. The full set of rules can be found in [4]. The core of our type system is a set of rules for reasoning about the typing judgment: $P;\ E;\ ls \vdash e : t$. P, the program being checked, is included here to provide information about class definitions. E is an environment providing types for the free variables of e. ls describes the set of locks that are statically known to be held when e is evaluated. t is the type of e.

The rule for accessing field $e.fd$ checks that e is a well-typed expression of some class type $cn\langle o_{1..n}\rangle$, where $o_{1..n}$ are actual owner parameters. It verifies that the class cn with formal parameters $f_{1..n}$ declares or inherits a field fd of type t and that the thread holds the lock on the root owner of e. Since t is declared inside the class, it might contain occurrences of this and the formal class parameters. When t is used outside the class, we rename this with the expression e, and the formal parameters with their corresponding actual parameters.

[EXPRESSION REFERENCE]

$$\frac{P;\ E;\ ls \vdash e : cn\langle o_{1..n}\rangle \qquad P \vdash (t\ fd) \in cn\langle f_{1..n}\rangle \qquad P;\ E \vdash \mathrm{RootOwner}(e) \in ls}{P;\ E;\ ls \vdash e.fd : t[e/\mathsf{this}][o_1/f_1]..[o_n/f_n]}$$

The rule for invoking a method checks that the arguments are of the right type and that the thread holds the locks on the root owners of all expressions in the accesses clause of the method. The expressions and types used inside the method are renamed appropriately when used outside their class.

[EXPRESSION INVOKE]

$$\frac{\begin{array}{c} P;\ E;\ ls \vdash e : cn\langle o_{1..n}\rangle \qquad P \vdash (t\ mn(t_j\ y_j\ ^{j\in 1..k})\ \mathsf{accesses}(e'^*)\ \{...\}) \in cn\langle f_{1..n}\rangle \\ P;\ E;\ ls \vdash e_j : t_j[e/\mathsf{this}][o_1/f_1]..[o_n/f_n] \\ P;\ E \vdash \mathrm{RootOwner}(e'_i[e/\mathsf{this}][o_1/f_1]..[o_n/f_n]) \in ls \end{array}}{P;\ E;\ ls \vdash e.mn(e_{1..k}) : t[e/\mathsf{this}][o_1/f_1]..[o_n/f_n]}$$

The rule for checking a method assumes that the locks on the root owners of all the expressions specified in the accesses clause are held. The rule then type checks the method body under this assumption.

[METHOD]

$$\frac{\begin{array}{c} E' = E,\ arg_{1..n} \qquad P;\ E' \vdash_{\mathsf{final}} e_i : t_i \qquad P;\ E' \vdash \mathrm{RootOwner}(e_i) = r_i \\ P;\ E';\ \mathsf{thisThread},\ r_{1..r} \vdash e : t \end{array}}{P;\ E \vdash t\ mn(arg_{1..n})\ \mathsf{accesses}(e_{1..r})\ \{e\}}$$

The rule for subtyping ensures that the parameters of the supertype are instantiated either with constants (self or thisThread) or with owners that are in scope, preserving the owner in the first position. The first owner must be preserved because the first owner in our system is special, in that it owns the this object.

[SUBTYPE]

$$\frac{\begin{array}{c} P;\ E \vdash cn_1\langle o_{1..n}\rangle \qquad P \vdash \mathsf{class}\ cn_1\langle f_{1..n}\rangle\ \mathsf{extends}\ cn_2\langle f_1\ o'^*\rangle\ \{...\} \\ \forall\ o'.\quad (o' = \mathsf{self}) \lor (o' = \mathsf{thisThread}) \lor (\exists\ j.\quad o' = f_j) \end{array}}{P;\ E \vdash cn_1\langle o_{1..n}\rangle <: cn_2\langle f_1\ o'^*\rangle\ [o_1/f_1]..[o_n/f_n]}$$

3 Safe Runtime Downcasts

This section describes how we support safe runtime downcasts efficiently. We describe our technique in the context of Mini Safe Concurrent Java (MSCJ) that we presented in Section 2. The type system for MSCJ described in Section 2 is a purely static type system. In fact, one way to compile and run a MSCJ program is to convert it into a Java program after type checking, by removing the type parameters and the `accesses` clauses from the program. The program can then be compiled and run as a regular Java program. However, a language like Java is not a purely statically typed language. Java allows downcasts that are checked at runtime. To support safe downcasts, the system must preserve some ownership information at runtime when ownership types are used in the context of a language like Java.

To express runtime casts in MSCJ, we extend the MSCJ grammar as follows.

$$e ::= \ldots \mid (cn\langle o_{1..n}\rangle)\ e$$

Fig. 6. Grammar Extensions to Support Runtime Casts

We next present the static type checking rules for runtime casts in MSCJ. Casting an object to a supertype of its declared type is always safe. Casting an object to a subtype of its declared type requires runtime checking. Section 2.2 contains the rule for subtyping.

[EXPRESSION UPCAST]

$$\frac{P;\, E;\, ls \vdash e : c_2 \qquad P;\, E \vdash c_2 <: c_1}{P;\, E;\, ls \vdash (c_1)\ e : c_1}$$

[EXPRESSION DOWNCAST (REQUIRES RUNTIME CHECK)]

$$\frac{P;\, E;\, ls \vdash e : c_1 \qquad P;\, E \vdash c_2 <: c_1}{P;\, E;\, ls \vdash (c_2)\ e : c_2}$$

To support downcasts, we store information on type parameters explicitly in objects and pass the information to code requiring the information. But if the system stores the owners of every object at runtime, this approach has the potential drawback of adding a per-object space overhead. Java objects are typically very small, so adding even a single field to every object may increase the size of most objects by a significant fraction. Our technique avoids the associated significant space overhead by storing runtime ownership information only for objects that are potentially involved in downcasts. Our technique for efficient implementation of type passing is based on two key observations about the nature of parameterization in ownership types.

The remainder of this section is organized as follows. Sections 3.1 and 3.2 describe the key observations that enable us to support downcasts efficiently. Sections 3.3 and 3.4 presents our technique for supporting safe downcasts.

```
 1   class T<thisOwner> {...}
 2   class TStack<thisOwner, TOwner> {...}
 3   class TStack2<thisOwner, TOwner> extends TStack<thisOwner, TOwner> {...}
 4
 5   Object<thisThread> o1;
 6   Object<thisThread> o2;
 7   Object<thisThread> o3;
 8   ...
 9   T<thisThread> t1;
10   T<self>       t2;
11   ...
12   TStack<thisThread, thisThread>  s1;
13   TStack<thisThread, self>        s2;
14   ...
15   TStack2<thisThread, thisThread> q1;
15   TStack2<thisThread, self>       q2;
16   ...
17   t1 = (T<thisThread>) o1;                   // Safe iff o1 belongs to class T
18   t2 = (T<self>)       o2;                   // Compile time error
19   ...
20   s1 = (TStack<thisThread, thisThread>)  o3; // Requires checking runtime ownership
21   s2 = (TStack<thisThread, self>)  o3;       // Requires checking runtime ownership
22   ...
23   q1 = (TStack2<thisThread, thisThread>) s1; // Safe iff s1 belongs to class TStack2
24   q2 = (TStack2<thisThread, self>) s1;       // Compile time error
```

Fig. 7. Runtime Downcasts

3.1 Downcasts to Types With Single Owners

A key observation that enables efficient implementation of downcasts is as follows. Consider the code in Figure 7. In Line 17, object `o1` of declared type $\mathsf{Object}\langle\mathsf{thisThread}\rangle$ is downcast to type $\mathsf{T}\langle\mathsf{thisThread}\rangle$. For this downcast, the owner of the declared type of `o1` matches the owner of the type that `o1` is being downcast into. Hence, this downcast is safe iff `o1` belongs to class T at runtime. It is unnecessary to check ownership information at runtime for this downcast.

In general, whenever an object is downcast into a type with matching owners, it is not necessary to check ownership information at runtime to ensure that the downcast is safe. If the owners of the declared type of the object match the owners of the type that the object is being downcast into, then the downcast will be safe iff the object belongs to the appropriate class at runtime (e.g., Lines 17 and 23 in Figure 7). If the owners do not match, the downcast will always fail (e.g., Line 18 and 24 in Figure 7).

The primary benefit of this observation is that whenever an object is downcast into a type with a single owner, it is unnecessary to check ownership information at runtime to ensure that the downcast is safe. Since a vast majority of classes in a system with ownership types have single owners, this implies that it is unnecessary to check ownership information at runtime for most of the downcasts. The only classes that usually have multiple owners are collection classes. The only times when it might be necessary to check ownership information at runtime to ensure that the downcast is safe is when an object is downcast into a type with multiple owners (e.g., Lines 20 and 21 in Figure 7).

3.2 Anonymous Owners

Another key observation that enables efficient implementation of downcasts is as follows. Consider the code in Figure 4. The TStack class in the figure is parameterized by thisOwner and TOwner. However, the owner parameter thisOwner is not used in the static scope where it is visible. Similarly, the owner parameter thisOwner for class T is not used in the body of class T. If an owner parameter is not used, it is unnecessary to name the parameter. Our system allows programmers to use ⟨-⟩ for such anonymous owner parameters. Figure 8 shows how we extend the MSCJ grammar to support anonymous owner parameters. Figure 11 shows the TStack example in Figure 4 implemented using anonymous owners for the TStack class and the T class.

$$defn ::= \ldots \mid \mathsf{class}\ cn\langle\text{-}\ f^*\rangle\ \mathsf{extends}\ c\ \{field^*\ meth^*\}$$

Fig. 8. Grammar Extensions to Support Anonymous Owners

The primary benefit of having anonymous owners is that if an owner parameter of a class is not named, it is unnecessary to store the owner parameter of the class at runtime, or pass the owner parameter to code that uses the class at runtime. In a system with ownership types, the only classes that usually have named owners are collection classes with multiple owners. Examples include Vector⟨-,elementOwner⟩, Hashtable⟨-,keyOwner,valueOwner⟩, etc. But most of the classes have single owners that are anonymous. It is unnecessary to store ownership information for those classes, or pass ownership information to code that uses those classes. Thus, our system incurs a runtime space and time overhead only for code that uses classes with named owner parameters like the collection classes. The rest of the code has no overhead in our system.

3.3 Preserving Ownership Information at Runtime

This section describes how our system preserves ownership information at runtime for classes with named owner parameters in the context of MSCJ. We presented the grammar for MSCJ in Figure 3 with extensions in Figures 6 and 8. This section presents the rules for translating a MSCJ program into an equivalent program in a Java-like language without ownership types. If we did not have to support safe runtime downcasts, the translation process would have been simple. We could have converted a MSCJ program into an equivalent Java-like program by simply removing the owner parameters and the accesses clauses. However, to support safe runtime downcasts, we must preserve some ownership information in the translation process.

The core of our translation is a set of rules of the form: $(\mathcal{T}[\![C]\!]\ P\ E) = C'$. The rule translates a code fragment C to a code fragment C'. P, the program being checked, is included here to provide information about class definitions. E

$$\begin{aligned}
(\mathcal{T}\llbracket P\rrbracket) &= (\mathcal{T}\llbracket defn^*\ e\rrbracket)\\
&= (\mathcal{T}\llbracket defn\rrbracket\ P)^*\ (\mathcal{T}\llbracket e\rrbracket\ P\ \emptyset)
\end{aligned}$$

$$\begin{aligned}
(\mathcal{T}\llbracket defn\rrbracket\ P) &= (\mathcal{T}\llbracket \mathsf{class}\ cn\langle f_{1..n}\rangle\ \mathsf{extends}\ cn'\langle o_{1..n'}\rangle\ \{field^*\ meth^*\}\rrbracket\ P)\\
&= \mathsf{class}\ cn\ \mathsf{extends}\ cn'\\
&\quad\ \{\mathsf{Object}\ \$f_{1..n}\ (\mathcal{T}\llbracket field\rrbracket\ P\ [f_{1..n}])^*\ (\mathcal{T}\llbracket method\rrbracket\ P\ [f_{1..n}])^*\}
\end{aligned}$$

$$\begin{aligned}
(\mathcal{T}\llbracket defn\rrbracket\ P) &= (\mathcal{T}\llbracket \mathsf{class}\ cn\langle\text{-}\ f_{2..n}\rangle\ \mathsf{extends}\ cn'\langle o_{1..n'}\rangle\ \{field^*\ meth^*\}\rrbracket\ P)\\
&= \mathsf{class}\ cn\ \mathsf{extends}\ cn'\\
&\quad\ \{\mathsf{Object}\ \$f_{2..n}\ (\mathcal{T}\llbracket field\rrbracket\ P\ [f_{2..n}])^*\ (\mathcal{T}\llbracket method\rrbracket\ P\ [f_{2..n}])^*\}
\end{aligned}$$

$$\begin{aligned}
(\mathcal{T}\llbracket meth\rrbracket\ P\ E) &= (\mathcal{T}\llbracket t\ mn(arg^*)\ \mathsf{accesses}\ (e_{\mathsf{final}}{}^*)\ \{e\}\rrbracket\ P\ E)\\
&= (\mathcal{T}\llbracket t\rrbracket\ P\ E)\ mn\ ((\mathcal{T}\llbracket arg\rrbracket\ P\ E)^*)\ \{(\mathcal{T}\llbracket e\rrbracket\ P\ E)\}
\end{aligned}$$

$$\begin{aligned}
(\mathcal{T}\llbracket field\rrbracket\ P\ E) &= (\mathcal{T}\llbracket[\mathsf{final}]_{\mathsf{opt}}\ t\ fd = e\rrbracket\ P\ E)\\
&= [\mathsf{final}]_{\mathsf{opt}}\ (\mathcal{T}\llbracket t\rrbracket\ P\ E)\ fd = (\mathcal{T}\llbracket e\rrbracket\ P\ E)
\end{aligned}$$

$$\begin{aligned}
(\mathcal{T}\llbracket arg\rrbracket\ P\ E) &= (\mathcal{T}\llbracket[\mathsf{final}]_{\mathsf{opt}}\ t\ fd\rrbracket\ P\ E)\\
&= [\mathsf{final}]_{\mathsf{opt}}\ (\mathcal{T}\llbracket t\rrbracket\ P\ E)\ fd
\end{aligned}$$

$$\begin{aligned}
(\mathcal{T}\llbracket t\rrbracket\ P\ E) &= (\mathcal{T}\llbracket cn\langle owner{+}\rangle\rrbracket\ P\ E)\\
&= cn
\end{aligned}$$

$$\begin{aligned}
(\mathcal{T}\llbracket t\rrbracket\ P\ E) &= (\mathcal{T}\llbracket \mathsf{int}\rrbracket\ P\ E)\\
&= \mathsf{int}
\end{aligned}$$

$$\begin{aligned}
(\mathcal{T}\llbracket e\rrbracket\ P\ E) &= (\mathcal{T}\llbracket (cn\langle o_{1..n}\rangle)\ e\rrbracket\ P\ E)\\
&= \{\$\mathrm{temp} = (cn)\ (\mathcal{T}\llbracket e\rrbracket\ P\ E);\\
&\quad\ \ \mathsf{if}\ (\$\mathrm{temp}.\$f_2\ \mathrel{!}= (\mathcal{O}\llbracket o_2\rrbracket\ P\ E))\ \mathsf{throw\ new\ ClassCastException};\\
&\quad\ \ ...;\\
&\quad\ \ \mathsf{if}\ (\$\mathrm{temp}.\$f_n\ \mathrel{!}= (\mathcal{O}\llbracket o_n\rrbracket\ P\ E))\ \mathsf{throw\ new\ ClassCastException};\\
&\quad\ \ \$\mathrm{temp}\}
\end{aligned}$$

$$\begin{aligned}
(\mathcal{T}\llbracket e\rrbracket\ P\ E) &= (\mathcal{T}\llbracket \mathsf{new}\ cn\langle o_{1..n}\rangle\rrbracket\ P\ E)\\
&= \{\$\mathrm{temp} = \mathsf{new}\ cn;\\
&\quad\ \ \$\mathrm{temp}.\$f_1 = (\mathcal{O}\llbracket o_1\rrbracket\ P\ E);\ ...;\ \$\mathrm{temp}.\$f_n = (\mathcal{O}\llbracket o_n\rrbracket\ P\ E);\\
&\quad\ \ \$\mathrm{temp}\}\\
&\\
&\quad\ \text{where } (\mathsf{class}\ cn\langle f_{1..n}\rangle\ ...) \in P
\end{aligned}$$

$$\begin{aligned}
(\mathcal{T}\llbracket e\rrbracket\ P\ E) &= (\mathcal{T}\llbracket \mathsf{new}\ cn\langle o_{1..n}\rangle\rrbracket\ P\ E)\\
&= \{\$\mathrm{temp} = \mathsf{new}\ cn;\\
&\quad\ \ \$\mathrm{temp}.\$f_2 = (\mathcal{O}\llbracket o_2\rrbracket\ P\ E);\ ...;\ \$\mathrm{temp}.\$f_n = (\mathcal{O}\llbracket o_n\rrbracket\ P\ E);\\
&\quad\ \ \$\mathrm{temp}\}\\
&\\
&\quad\ \text{where } (\mathsf{class}\ cn\langle\text{-}\ f_{2..n}\rangle\ ...) \in P
\end{aligned}$$

$$\begin{aligned}
(\mathcal{O}\llbracket o\rrbracket\ P\ E) &= (\mathcal{O}\llbracket \mathsf{thisThread}\rrbracket\ P\ E) && = \$\mathrm{Owner.THISTHREAD()}\\
(\mathcal{O}\llbracket o\rrbracket\ P\ E) &= (\mathcal{O}\llbracket \mathsf{self}\rrbracket\ P\ E) && = \$\mathrm{Owner.SELF()}\\
(\mathcal{O}\llbracket o\rrbracket\ P\ E) &= (\mathcal{O}\llbracket f\rrbracket\ P\ [...\ f\ ...]) && = \$\mathrm{f}\\
(\mathcal{O}\llbracket o\rrbracket\ P\ E) &= (\mathcal{O}\llbracket \mathrm{e}\rrbracket\ P\ E) && = (\mathcal{T}\llbracket \mathrm{e}\rrbracket\ P\ E)
\end{aligned}$$

$$\begin{aligned}
(\mathcal{T}\llbracket e\rrbracket\ P\ E) &= (\mathcal{T}\llbracket x\rrbracket\ P\ E) && = x\\
(\mathcal{T}\llbracket e\rrbracket\ P\ E) &= (\mathcal{T}\llbracket x = e\rrbracket\ P\ E) && = x = (\mathcal{T}\llbracket e\rrbracket\ P\ E)\\
(\mathcal{T}\llbracket e\rrbracket\ P\ E) &= (\mathcal{T}\llbracket e.fd\rrbracket\ P\ E) && = (\mathcal{T}\llbracket e\rrbracket\ P\ E).fd\\
(\mathcal{T}\llbracket e\rrbracket\ P\ E) &= (\mathcal{T}\llbracket e_1.fd = e\rrbracket\ P\ E) && = (\mathcal{T}\llbracket e_1\rrbracket\ P\ E).fd = (\mathcal{T}\llbracket e\rrbracket\ P\ E)\\
(\mathcal{T}\llbracket e\rrbracket\ P\ E) &= (\mathcal{T}\llbracket e_1.mn(e^*)\rrbracket\ P\ E) && = (\mathcal{T}\llbracket e_1\rrbracket\ P\ E).mn((\mathcal{T}\llbracket e\rrbracket\ P\ E)^*)\\
(\mathcal{T}\llbracket e\rrbracket\ P\ E) &= (\mathcal{T}\llbracket e_1;e_2\rrbracket\ P\ E) && = (\mathcal{T}\llbracket e_1\rrbracket\ P\ E);(\mathcal{T}\llbracket e_2\rrbracket\ P\ E)\\
(\mathcal{T}\llbracket e\rrbracket\ P\ E) &= (\mathcal{T}\llbracket \mathsf{let}\ (arg{=}e_1)\ \mathsf{in}\ \{e\}\rrbracket\ P\ E) && = \mathsf{let}\ (arg{=}(\mathcal{T}\llbracket e_1\rrbracket\ P\ E))\ \mathsf{in}\ \{(\mathcal{T}\llbracket e\rrbracket\ P\ E[arg])\}\\
(\mathcal{T}\llbracket e\rrbracket\ P\ E) &= (\mathcal{T}\llbracket \mathsf{synchronized}\ (e_1)\ \mathsf{in}\ \{e\}\rrbracket\ P\ E) && = \mathsf{synchronized}\ ((\mathcal{T}\llbracket e_1\rrbracket\ P\ E))\ \mathsf{in}\ \{(\mathcal{T}\llbracket e\rrbracket\ P\ E)\}\\
(\mathcal{T}\llbracket e\rrbracket\ P\ E) &= (\mathcal{T}\llbracket \mathsf{fork}\ (x^*)\ \{e\}\rrbracket\ P\ E) && = \mathsf{fork}\ (x^*)\ \{(\mathcal{T}\llbracket e\rrbracket\ P\ E)\}
\end{aligned}$$

Fig. 9. Translation Function

```
public class $Owner {
  public static Object self = "self";

  public static Object SELF() { return self; }
  public static Object THISTHREAD() { return Thread.currentThread(); }
}
```

Fig. 10. The $Owner Class

is an environment containing the formal owner parameters in scope in C. The translated code uses the $Owner class shown in Figure 10. The $Owner class contains two static methods that return objects that represent the thisThread owner and the self owner respectively. The translation rules are presented in Figure 9. Section 3.4 explains the translation process with examples.

3.4 Implementation

This section illustrates with examples how our implementation preserves ownership information at runtime for classes with named owner parameters. If a Safe Concurrent Java (SCJ) program is well-typed with respect to the rules for static type checking, our implementation translates the program into an equivalent Java program. (Actually, our implementation translates a SCJ program into Java bytecodes directly. But for ease of presentation, we will describe an equivalent translation into Java code.) The translation mechanism is illustrated in Figures 11, 12, 13, and 14. Figure 11 shows a TStack class with anonymous owners. Figure 12 shows client code that uses the TStack class. Figures 13 and 14 show the translation of the TStack code and the client code.

Classes Classes in the translated code contain extra owner fields, one for each named owner parameter of the class. For example, in Figure 13, the translated TStack class has an extra $TOwner field. The translated TNode class has two extra fields: $thisOwner and $TOwner. The translated T class has no extra fields since the T class does not have any named owner parameters.

Constructors Constructors in the translated code contain extra owner arguments, one for each named owner parameter of the class. The constructors in the translated code initialize the owner fields of the class with the owner arguments of the constructor. For example, in Figure 13, the constructor for TStack has an extra $TOwner argument. The constructor initializes the $TOwner field of the TStack object from the $TOwner argument.

Allocation Sites Client code in the translated version that creates a new object of a class must pass extra owner arguments to the constructor, one for each named owner parameter of the class. If the owner is an expression that evaluates to an object, the client code passes the object to the constructor. For example, in Figure 13, the push method in TStack passes the this object as the

```
// TStack has an anonymous owner, TOwner owns the T objects in the stack.

class TStack<-, TOwner> {

  TNode<this, TOwner> head = null;

  TStack() {}
  void push(T<TOwner> value) accesses (this) {
    TNode<this, TOwner> newNode = new TNode<this, TOwner>(value, head); head = newNode;
  }
  T<TOwner> pop() accesses (this) {
    T<TOwner> value = head.value(); head = head.next(); return value;
  }
}

class TNode<thisOwner, TOwner> {

  T<TOwner> value; TNode<thisOwner, TOwner> next;

  TNode(T<TOwner> v, TNode<thisOwner, TOwner> n) accesses (this) {
    this.value = v; this.next  = n;
  }
  T<TOwner> value() accesses (this) { return value; }
  TNode<thisOwner, TOwner> next() accesses (this) { return next; }
}

class T<-> { int x=0; }
```

Fig. 11. TStack With Anonymous Owners

```
class T<-> {...}
class TStack<-, TOwner> {...}
class TStack2<-, TOwner> extends TStack<-, TOwner> {...}

Object<thisThread> o1;
Object<thisThread> o2;
...
TStack<thisThread, thisThread>  s1 = new TStack<thisThread, thisThread>;
TStack<thisThread, self>        s2 = new TStack<thisThread, self>;
...
TStack2<thisThread, thisThread> q1;
TStack2<thisThread, self>       q2;
...
s1 = (TStack<thisThread, thisThread>)  o1;
s2 = (TStack<thisThread, self>)  o2;
...
q1 = (TStack2<thisThread, thisThread>) s1;
q2 = (TStack2<thisThread, self>) s2;
...
boolean b1 = (o1 instanceof TStack<thisThread, thisThread>);
boolean b2 = (o2 instanceof TStack<thisThread, self>);
```

Fig. 12. Client Code for TStack

```
// TStack has an anonymous owner, TOwner owns the T objects in the stack.

class TStack {
  Object $TOwner;

  TNode head = null;

  TStack(Object $TOwner) {
    this.$TOwner = $TOwner;
  }
  void push(T value) {
    TNode newNode = new TNode(this, $TOwner, value, head); head = newNode;
  }
  T pop() {
    T value = head.value(); head = head.next(); return value;
  }
}

class TNode {
  Object $thisOwner, $TOwner;

  T value; TNode next;

  TNode(Object $thisOwner, Object $TOwner, T v, TNode n) {
    this.$thisOwner = $thisOwner; this.$TOwner = $TOwner;
    this.value = v; this.next  = n;
  }
  T value() { return value; }
  TNode next() { return next; }
}

class T { int x=0; }
```

Fig. 13. Translation of TStack in Figure 11

```
class T {...}
class TStack {...}
class TStack2 extends TStack {...}

Object o1;
Object o2;
...
TStack s1 = new TStack($Owner.THISTHREAD());
TStack s2 = new TStack($Owner.SELF());
...
TStack2 q1;
TStack2 q2;
...

s1 = (TStack) o1;
if (s1.$TOwner != $Owner.THISTHREAD()) throw new ClassCastException();
s2 = (TStack) o2;
if (s2.$TOwner != $Owner.SELF()) throw new ClassCastException();
q1 = (TStack2) s1;
if (q1.$TOwner != $Owner.THISTHREAD()) throw new ClassCastException();
q2 = (TStack2) s2;
if (q2.$TOwner != $Owner.SELF()) throw new ClassCastException();
...
boolean b1 = ((o1 instanceof TStack) && (((TStack) o1).$TOwner == $Owner.THISTHREAD()));
boolean b2 = ((o2 instanceof TStack) && (((TStack) o2).$TOwner == $Owner.SELF()));
```

Fig. 14. Translation of TStack Client Code in Figure 12

first argument to the TNode constructor. If the owner is a formal parameter, the client code passes the value of the formal parameter stored in one of its extra owner fields. For example, in Figure 13, the push method in TStack passes the value stored in the $TOwner field as the second argument to the TNode constructor. If the owner is thisThread or self, the client code passes the object returned by $Owner.THISTHREAD() or $Owner.SELF() to the constructor. For example, in Figure 14, the client code creates TStacks s1 and s2 by passing $Owner.THISTHREAD() and $Owner.SELF() to the TStack constructor respectively.

Casts Casts in the translated code not only check that the Java types match, but also check that the owners match. For example, in Figure 14, in Line 15, the translated code not only checks that o1 is of Java type TStack, but also checks that the owner of the T elements in the TStack is thisThread. In Line 16, the translated code not only checks that o2 is of Java type TStack, but also checks that the owner of the T elements in the TStack is self.

InstanceOf The instanceof operation in the translated code returns true iff the Java types match and the owners match. For example, in Figure 14, in Line 20, the instanceof operation returns true iff o1 is of Java type TStack and the owner of the T elements in the TStack is thisThread. In Line 21, the instanceof operation returns true iff o2 is of Java type TStack and the owner of the T elements in the TStack is self.

Parameterized Methods Parameterized methods are handled similar to parameterized classes. For ease of presentation, the MSCJ language we described in Section 2 has only parameterized classes but not parameterized methods. But our implementation handles both parameterized classes and parameterized methods. Named owner parameters of methods are explicitly passed as arguments to the methods in the translated code.

4 Related Work

4.1 Ownership Types

Ownership types were first introduced in Flexible Alias Protection [9], and formalized in [8]. Parameterized Race Free Java (PRFJ) [5] extends ownership types to support inheritance and dynamic aliases (which allow, e.g., the description of iterators) and uses them to statically ensure the absence of data races in Java programs. PRFJ also uses effects clauses and combines ownership types with unique pointers [16], and read-only fields and objects, which allows many important idioms to be expressed. Safe Concurrent Java (SCJ) [4] extends PRFJ to prevent both data races and deadlocks in Java programs. AliasJava [2] uses ownership types to aid software evolution. Cyclone [13] uses a similar type system to guarantee absence of memory errors in programs with explicit deallocation.

4.2 Parametric Polymorphism in Java

Our implementation of parameterized ownership types is related to the *type passing* approach [17, 19, 18] of implementing parametric polymorphism in Java. In the type passing approach, information on type parameters is explicitly stored in objects and passed to code requiring them. But if the system stores the owners of every object at runtime, this approach has the potential drawback of adding a per-object space overhead. Java objects are typically very small, so adding even a single field to every object may increase the size of most objects by a significant fraction. This paper describes an efficient technique for implementing ownership types using the type passing approach that avoids the associated significant space overhead by storing runtime ownership information only for some objects.

4.3 Types for Safe Concurrent Programing

Many researchers have proposed language mechanisms for safe concurrent programming. The Extended Static Checker for Java (Esc/Java) [10] is an annotation based system that uses a theorem prover to statically detect many kinds of errors including data races and deadlocks. Race Free Java [11] extends the static annotations in Esc/Java into a formal race-free type system. Guava [3] is another dialect of Java for preventing data races. Parameterized Race Free Java (PRFJ) [5] builds on Race Free Java and lets programmers write generic code to implement a class, and create different objects of the same class that have different protection mechanisms. PRFJ also supports objects with unique pointers and read-only objects and fields that can be accessed without synchronization. Safe Concurrent Java (SCJ) [4] extends PRFJ to prevent both data races and deadlocks in multithreaded programs.

5 Conclusions

The possibility of aliasing between objects constitutes one of the primary challenges in understanding and reasoning about correctness of object-oriented programs. Ownership types provide a principled way of specifying statically enforcable restrictions on object aliasing. Ownership types have been used to aid program understanding and evolution, verify absence of data races and deadlocks in multithreaded programs, and verify absence of memory errors in programs with explicit deallocation.

This paper describes an efficient technique for supporting safe runtime downcasts with ownership types. This technique uses the type passing approach, but avoids the associated significant space overhead by storing runtime ownership information only for objects that are potentially involved in downcasts. Moreover, this technique does not use any inter-procedural analysis, so it preserves the separate compilation model of Java. We implemented our technique in the context of Safe Concurrent Java, which is an extension to Java that uses ownership types to guarantee the absence of data races and deadlocks in well-typed programs. Our approach is JVM-compatible: our implementation translates programs to bytecodes that can be run on regular JVMs.

References

1. O. Agesen, S. N. Freund, and J. C. Mitchell. Adding type parameterization to the Java language. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1997.
2. J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.
3. D. F. Bacon, R. E. Strom, and A. Tarafdar. Guava: A dialect of Java without data races. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2000.
4. C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.
5. C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2001.
6. G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1998.
7. R. Cartwright and G. Steele. Compatible genericity with run-time types for the Java programming language. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1998.
8. D. G. Clarke, J. Noble, and J. M. Potter. Simple ownership types for object containment. In *European Conference for Object-Oriented Programming (ECOOP)*, June 2001.
9. D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1998.
10. D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, 1998.
11. C. Flanagan and S. N. Freund. Type-based race detection for Java. In *Programming Language Design and Implementation (PLDI)*, June 2000.
12. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.
13. D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Programming Language Design and Implementation (PLDI)*, June 2001.
14. J. Hogg, D. Lea, alan Wills, and D. de Champeaux. The Geneva convention on the treatment of object aliasing, April 1992.
15. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1997.
16. N. Minsky. Towards alias-free pointers. In *European Conference for Object-Oriented Programming (ECOOP)*, July 1996.
17. A. C. Myers, J. A. Bank, and B. Liskov. Parameterized types for Java. In *Principles of Programming Languages (POPL)*, January 1997.
18. M. Viroli. Parametric polymorphism in Java: An efficient implementation for parametric methods. In *Symposium on Applied Computing (SAC)*, March 2001.
19. M. Viroli and A. Natali. Parametric polymorphism in Java: An approach to translation based on reflective features. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2000.