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. This paper describes an efficient technique for supporting safe runtime downcasts in a system with ownership types. This technique uses the type passing approach, but avoids the associated significant space overhead by storing only the runtime ownership information that is potentially needed to support safe 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 statically guarantee the absence of data races and deadlocks. Our approach is JVM-compatible: our implementation translates programs to bytecodes that can be run on regular JVMs.

1 Introduction

Ownership types [4, 5, 7, 12, 13] provide a statically enforceable way of specifying object encapsulation. The idea is that an object can *own* subobjects that it depends on, thus preventing them from being accessible outside. Object encapsulation enables local reasoning about program correctness in object-oriented programs. Ownership-based type systems have also been used for preventing data races [7] and deadlocks [4] in multithreaded programs, for preventing memory errors in programs that use region-based memory management [8], for supporting modular software upgrades in persistent object stores [6], for modular specification of effects clauses in the presence of subtyping [5, 7, 12] (so that they can be used as an alternative to data groups [16]), and for program understanding [2].

In ownership type systems, programmers 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 [1, 9, 18, 20]. 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 [7]), but it is usually unnecessary to preserve the ownership information at runtime. However, languages like Java [15] 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 for implementing parametric polymorphism in a language like Java. The *type erasure* approach [9, 10] 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 [18, 20, 19], 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 small, so adding even a single field to every object increases 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 only the runtime ownership information that is potentially needed to support safe 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, 7], 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 [17].

We note that a similar approach has been used in [2] to implement safe runtime downcasts with ownership types.

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

2 Mini 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 if x owns y. Our type system statically verifies that a program respects the ownership properties shown in Figure 2.

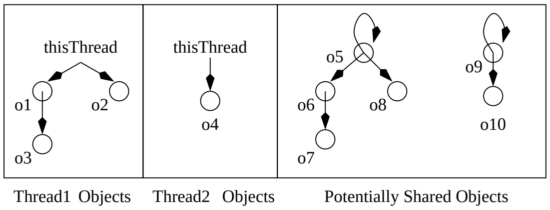

Fig. 1. An Ownership Relation

1. The owner of an object does not change over time.
2. The ownership relation forms a forest of rooted trees. The roots can have self loops.
3. To safely access an object, a thread must hold the lock on the *root owner* of that object (the root of the ownership tree that the object belongs to).
4. Every thread implicitly holds the lock on its corresponding thisThread owner. A thread can thus access objects owned by its thisThread without 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 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

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

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

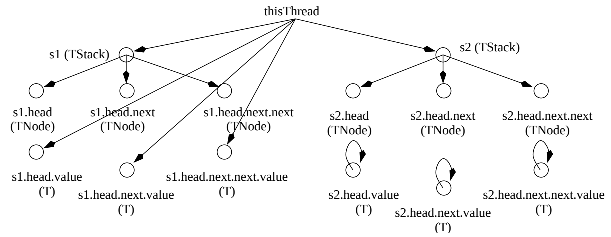

Fig. 5. Ownership Relation for TStacks s1 and s2

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.

2.1 Static Type Checking

This section describes some of the important type checking rules. 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]

$$\mathrm{Renamed}(\alpha) \overset{\mathrm{def}}{=} \alpha[e/\mathsf{this}][o_1/f_1]..[o_n/f_n][e_1/y_1]..[e_k/y_k]$$

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

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 subtyping rule 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]

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

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).

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. 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, 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 casts. Casting an object to a supertype of its declared type is always safe. Casting to a subtype of the declared type requires runtime checking. Section 2.1 contains the subtyping rule.

[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 small, so adding even a single field to every object increases the size of most objects by a significant fraction. Our technique avoids the associated significant space overhead by storing only the runtime ownership information that is potentially needed to support safe downcasts. Our technique 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 present our technique for supporting safe 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 16, object `o1` of declared type `Object⟨thisThread⟩` is downcast to type `T⟨thisThread⟩`. For this downcast, the owner of the declared type of `o1` matches the owner of the type that `o1` is being

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

Fig. 7. Runtime Downcasts

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, for any subtype declaration where all the formal owner parameters in the subtype are included in the supertype, it is not necessary to check ownership information at runtime when an object is downcast from the supertype to the subtype. If the owners of the supertype match the owners of the subtype, then the downcast will be safe iff the object belongs to the appropriate class at runtime (e.g., Lines 16 and 22 in Figure 7). If the owners do not match, the downcast will always fail (e.g., Lines 17 and 23 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 19 and 20 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.

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

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 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 is an environment containing the formal owner parameters in scope in C. The translated code uses the $Owner class shown in Figure 9. The $Owner class

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

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

Fig. 9. The $Owner Class

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 10. Translation Function

```
// 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

contains two static methods that return objects that represent the `thisThread` owner and the `self` owner respectively. The translation rules are presented in Figure 10. 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. 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 Allocation sites in the translated code must pass extra owner arguments to constructors, one for each named owner parameter of the corresponding 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 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, instanceof returns true iff o1 is of Java type TStack and the owner of the T elements in the TStack is thisThread. In Line 21, instanceof returns true iff o2 is of Java type TStack and the owner of the T elements in the TStack is self.

Arrays The technique described in this paper does not support safe runtime downcasts to array types. This is because we cannot add extra owner fields to array objects in the translated code and yet remain JVM-compatible. If a programmer wants to downcast from java.lang.Object to an array type in our system, the programmer can create a wrapper object that contains the array object and perform the downcast on the wrapper object.

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 Conclusions

This paper describes an efficient technique for supporting safe runtime downcasts in a system with ownership types. This technique uses the type passing approach, but avoids the associated significant space overhead by storing only the runtime ownership information that is potentially needed to support safe downcasts. The technique preserves the separate compilation model of Java and is JVM-compatible: it translates programs to bytecodes that can be run on regular JVMs.

Acknowledgments

This research was supported by DARPA/AFRL Contract F33615-00-C-1692, NSF Grant CCR00-86154, NSF Grant CCR00-73513, and the Singapore-MIT Alliance.

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. C. Boyapati, R. Lee, and M. Rinard. Safe runtime downcasts with ownership types. Technical Report TR-853, MIT Laboratory for Computer Science, June 2002.
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, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *Principles of Programming Languages (POPL)*, January 2003.
6. C. Boyapati, B. Liskov, L. Shrira, C. Moh, and S. Richman. Lazy modular upgrades in persistent object stores. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2003.
7. 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.
8. C. Boyapati, A. Salcianu, W. Beebee, Jr., and M. Rinard. Ownership types for safe region-based memory management in Real-Time Java. In *Programming Language Design and Implementation (PLDI)*, June 2003.
9. 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.
10. 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.
11. D. G. Clarke. Object ownership and containment. PhD thesis, University of New South Wales, Australia, July 2001.
12. D. G. Clarke and S. Drossopoulou. Ownership, encapsulation and disjointness of type and effect. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.
13. 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.
14. D. G. Clarke and T. Wrigstad. External uniqueness is unique enough. In *European Conference for Object-Oriented Programming (ECOOP)*, July 2003.
15. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.
16. K. R. M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects. In *Programming Language Design and Implementation (PLDI)*, June 2002.
17. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1997.
18. A. C. Myers, J. A. Bank, and B. Liskov. Parameterized types for Java. In *Principles of Programming Languages (POPL)*, January 1997.
19. M. Viroli. Parametric polymorphism in Java: An efficient implementation for parametric methods. In *Symposium on Applied Computing (SAC)*, March 2001.
20. 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.