Using ownership types to support library aliasing boundaries

Luke Wagner Jaakko Järvi Bjarne Stroustrup
Texas A&M University
{lw,jarvi,bs}@cs.tamu.edu

Abstract

This paper describes a library for concurrency used in a 10-developer videogame project. The developers were inexperienced, yet there were no problems with data races in the multi-threaded application. We credit this to the explicit representation of ownership in the design of the library. Correct library usage implies aliasing boundaries which bear a strong resemblance to the owners-as-dominators property enforced by ownership types. We explore other situations where analogous aliasing boundaries exist and discuss a family of related libraries that could benefit from a design explicitly representing ownership. The ownership relations in the library currently have no support from the type system. We examine approaches to embed static checking of the aliasing boundaries in our implementation language, C++.

Categories and Subject Descriptors D.1.3 [*Software*]: Programming Techniques—Concurrent Programming; D.3.3 [*Software*]: Language Constructs and Features—Control structures

General Terms Design

Keywords Ownership types, C++, Data Races, Concurrency

1. Introduction

In research on type systems for object-oriented languages, an important property of interest is local reasoning. The challenge lies in the fact that an object's state is comprised not only of its immediate data members, but also the transitive closure of all the states of the objects on which it depends [1]. To provide a "deeper" form of encapsulation than directly supported by current languages, ownership types [2, 3] allow a class to identify its dependencies on other objects and then prevent outsiders from acquiring references to those dependencies. With these limitations on aliasing, it is possible to reason about the correctness of a class by looking only at the code for that class and its dependencies.

However, local reasoning for the programmer is not the only benefit from using aliasing boundaries. Researchers have demonstrated that higher-level program guarantees can be made by building on ownership type systems [4–8]. This paper presents an additional example where aliasing boundaries in a program can be beneficial: a library for concurrency developed and successfully used in a large student videogame project. We show that the aliasing boundaries required for correct library usage strongly resemble the owners-as-dominators property enforced on an object graph by ownership types [2]. Based on this, the paper presents a method by which code using the library could be checked using ownership types.

Based on the positive experience with the concurrency library, this paper considers a family of related libraries that could benefit from a similar approach. Together, these libraries can be seen as the decomposition of the separation facilities built into a traditional process, so that each individual separation facility may be applied at the sub-process level.

In sum, this paper is an experience report and a position paper that (1) describes a set of library abstractions and programming conventions that restrict aliasing in order to guarantee the absence of data-races; (2) identifies the correspondence of the aliasing boundaries required by the library with those expressed with ownership types; (3) describes the type system extensions necessary to move from a documented usage rule to statically ensuring that aliasing boundaries are respected; (4) outlines an economical implementation approach to embed those extensions into standard C++, by inspecting the program's AST with an extended type checker; and (5) identifies the aliasing boundaries discussed as a general pattern for a related class of libraries, justifying the effort to develop the checking mechanisms.

This paper is organized as follows. Section 2 describes the concurrency library and Section 3 how ownership types can be used to check its correct use. Section 5 discusses the C++ embedding, and the other libraries that could benefit from the same technique. Section 6 mentions related work. Section 7 concludes and discusses future plans.

2. The Library

This section describes a simple concurrency library that was developed for a videogame project written in C++. The game is called "...and then the World was Consumed by Monsters" and can be downloaded from the development team's website [9]. The project, organized by the Texas Aggie Game Developers, included 10 undergraduate student developers over a period of 6 months with no other experienced oversight. Thus, simplicity and understandability were key to the success of the project.

In the videogame development community, amateurs are often strongly discouraged from using concurrency by the more experienced because of the difficult class of bugs it can introduce. However, several game constraints made it necessary to offload computation and blocking API calls to other threads. First, as with most interactive videogames, there is an underlying rendering loop which repaints the screen. To maintain a visually smooth animation, each frame should take less than 30 ms. Second, the game allows the user to control a character that roams around a virtual world. The representation of the virtual world can be much larger than what fits in memory. This requires the world to be cut into smaller chunks which contain all the geometry, collision data, and creatures for a small area of the world. As the user moves into new areas, chunks get loaded and dropped, which requires I/O operations to load the memory, OS, and graphics resources for those chunks. Since some of these operations do not provide an asynchronous option and can have a high latency, most videogames either try to perform them all at once before the game starts or batch the operations and stall the user at chosen points when executing the batch. Such stalls did not fit nicely into the gameplay, so a separate thread was needed to handle concurrent world loading.

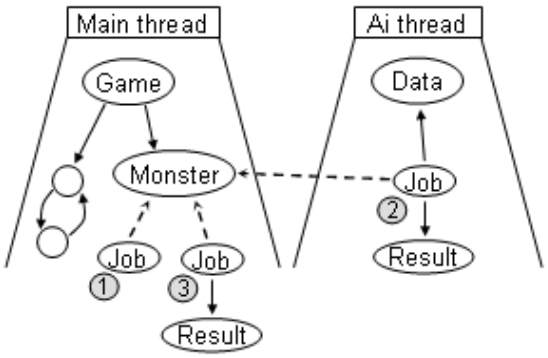

Figure 1. Travelling object with a tether, demonstrating the three states 1–3 of the Job object.

The same mechanism used for loading was later expanded to simplify AI programming. Here, the problem is a possibly expensive algorithm to find a path between two points. While the algorithm could be extended to moderate its execution time, a simpler approach is sending requests for computation to another thread with a lower priority. Thus, for slower machines where rendering takes a greater percentage of the time, enemies will simply take longer to decide where to go instead of hurting the framerate. An obvious additional benefit is the ability to utilize multiple hardware threads.

To avoid requiring the rest of the program to deal with shared mutable data and locking, the model for concurrency was based on the communicating sequential processes (CSP) metaphor (where CSP processes are just threads and the shared address space is not explicitly used). When extended to object-oriented programming, all objects are understood to belong to a single thread and messages between threads take the form of objects that can dynamically change membership. Objects that are allowed to change membership are called *travelling objects* and all others are called *local objects*.

A common need in the project was for a travelling object to create a reference to a local object, travel to a different thread to do some work, and then return to the original thread to use the reference. The synchronization implied by travelling prevents such behavior from being a data race. However, the situation is complicated by the fact that the local object may be destroyed while the travelling object is away. In a single-threaded scenario, a *weak pointer* library primitive comparable, e.g., to Boost weak pointer [10], is used when the pointee is allowed to be destroyed while another objects points to it. Although, the weak pointer implementation could have been extended to be made thread-safe, at the expense of synchronization overhead for all operations, this is more powerful than is necessary: a weak pointer maintains the liveness of the local object while the travelling object is in other threads. All a travelling object needs is to discover, when it returns, if the pointee has been destroyed in the interim.

To address the need for a simplified cross-thread weak pointer, the thread library provides a new, thread-aware smart pointer called a *tether*. Taking advantage of a tether's restricted semantics, the library is able to use the synchronization points already in place for transferring travelling objects between threads to keep the tethers coherent when they change threads.

Figure 1 demonstrates a typical usage scenario for a tether. In the figure: labels 1-3 show three steps in execution, the solid arrows represent normal references, and the dashed arrows represent tethers. In this scenario, a **Monster** local object needs a path in order to attack the player. Since the path finding algorithm should not be run in the rendering thread, the **Monster** creates a **Job** object (1) and ships it off to do the work in another thread. Before leaving, the **Job** creates a tether to the **Monster**. Next, the **Job** arrives (2) and is given a temporary local reference to the **Data** object which it can use to do the path finding computation. While in a different thread than the **Monster**, the tether held by the **Job** cannot be dereferenced. After **Job** finishes and returns to the original thread (3), it uses the tether to check whether the tethered **Monster** is still alive, and if so, the **Job** hands over the computation results.

Although more general usage of travelling objects could be supported using these library metaphors, the functionality required by the project only needed threads to act like assembly lines which processed jobs FIFO in the manner just described. Accordingly, **AssemblyLine** is the library primitive for creating such threads:

```
template <class GenericHost>
class AssemblyLine {
  GenericHost *host;
public:
  ...
  void send(typename GenericHost::Guest *);
  void receive_returning();
};
```

Since **AssemblyLine** does not know what to do with travelling objects, it is parametrized by a **GenericHost**. The host's responsibility is to receive incoming travelling objects and to provide them access to the necessary local data structures. Additionally, the type of travelling objects is determined by the **Guest** associated type of **GenericHost**. After starting a new OS thread in its constructor, an **AssemblyLine** will create an instance of **GenericHost**, which will be the first client object local to the new thread. To allow returning travelling objects to reenter the thread, the main thread synchronizes with **AssemblyLine** by calling **receive_returning()**.

To give a better idea of the library's use, we now walk through some skeleton code using the library in the path finding scenario. At the top-level of the application, a **Game** is created which, in turn, creates an **AssemblyLine**:

```
class Game {
  AssemblyLine<Host> ai_thread;
  ...
public:
  void run() {
    while (!quit) {
      ...
      ai_thread.receive_returning();
    }
  }
};
int main() {
  Game g;
  g.run();
}
```

When **ai_thread** is destroyed by **Game**, job processing will be stopped, all pending jobs will be deleted, and the OS thread will be released. **Host** parametrizes **AssemblyLine** and holds the path finding data that is needed by **Job**s:

```
class Host {
  Data data;
public:
  typedef Job Guest;
  void arrived(Job &guest) {
    guest.do_work(data);
  }
};
```

When a travelling object is sent from the main thread and gets pulled off the queue by **AssemblyLine**'s thread, it is handed over

to **Host** by calling **arrived()**. When **arrived()** returns, **AssemblyLine** will send the travelling object back to the main thread. In addition to being compatible with **Host::Guest**, a travelling object's class must inherit from the **TravellerBase** library base class:

```
class Job : public TravellerBase {
  Tether<Monster> tether;
  ...
public:
  Job(Monster &m, ...) : tether(create_tether(m)) { ... }
  void do_work(Data &);
  void welcome_back() {
    if (tether)
      tether->found_path(...);
  }
};
```

Travelling objects can choose to have any number of tethers to local objects using the **Tether** template class, parametrized by the type of the pointee. **Tether** follows the C++ smart pointer idiom and guards access to the pointee through **operator->()**. For **AssemblyLine** and **Tether** to cooperate in keeping the tether coherent when it changes threads, construction of **Tether** is abstracted by the **create_tether()** protected member function inherited from **TravellerBase**. When a travelling object is accepted back into the main thread, **welcome_back()** is called. **Job** can then safely use its **Tether** after testing that the **Monster** object it was pointing to has not been destroyed.

Finally, using **Job** in **Monster** is fairly simple:

```
class Monster {
  AssemblyLine<Host> &ai_thread;
  ...
public:
  ...
  void think() {
    if (... I want to attack ...)
      ai_thread.send(new Job(*this, ...));
  }
  void found_path(...);
};
```

The project did not need **Job** objects after they returned to the main thread, so the **AssemblyLine** takes the liberty of deleting them. Altogether, the end-to-end order of function calls corresponding to Figure 1 is: **Monster::think()**, **Job::Job()**, **AssemblyLine::send()**, **Host::arrived()**, **Job::do_work()**, **AssemblyLine::receive_returning()**, **Job::welcome_back()**, **Monster::found_path()**, **Job::~Job()**.

Although message-based schemes are often viewed as more complex than shared-memory schemes when used for low level parallel programming, as used in the videogame project for simple task-level parallelism, we found the message-passing approach to be a clear mental model of concurrent execution for the programmer compared to shared memory with locking. Programming with this model, we did not experience data races. This could be attributed to the smaller scale of the student project, or the fear of concurrency imbued in the team by horror stories, but we believe the library design was an important part.

3. Checking Usage

The library described in Section 2 helps programmers by providing a simple mental model and set of tools for programming concurrency. This section describes how the type system could be enlisted to help as well. What is described is a correspondence between ownership typing judgements and aliasing restrictions in the concurrency library. The code shown is what the ownership type system needs to see, not what needs to be written in the actual C++ code. A lightweight embedding in C++ is discussed in Section 4. The syntax used to express the ownership typing concepts is based on Joline [11] and Ownership Generic Java (OGJ) [3]. In some places, features of C++ will be mixed in where they are needed by the library.

Another point to clarify is the meaning of *ownership*. Ownership types are traditionally presented in the context of a language with garbage collection and so the main issue is accessibility. However, in the context of C++, ownership can also refer to the responsibility of an object to manage the lifetime of the resources it owns. This paper limits its discussion of ownership to issues of accessibility; static guarantees involving object lifetimes are not addressed.

This section first discusses the basics of ownership types and then describes how they can be used by each piece of the library.

3.1 Background

Ownership types can be used to statically limit what references are allowed between objects. Considering objects and their references as a graph, ownership types allow the user to draw boundaries around parts of the graph, limiting incoming references. What follows is a brief explanation of how this is accomplished. Although it sounds like extra runtime state and checking is being added, none of it is needed after type checking; the runtime behavior of the program is not modified.

First, every object is given a unique *ownership context*. An ownership context can be thought of as a value of an opaque type. The only purpose of an ownership context is to be part of the type of an object. An object's class is augmented to take, as a generic parameter, the ownership context of some other object, which becomes its owner. Because ownership contexts are values, this creates a relation between objects, not types. Additionally, there is an omnipresent, disembodied **world** ownership context which is not associated with any object. Because an owner has to be constructed before the objects it owns, ownership is acyclic. Furthermore, all objects have exactly one owner, so the ownership relation forms a tree rooted at **world**.

For an object to hold or use a reference to another object, static type checking demands that the reference have a type. Ownership types limit aliasing by controlling what types can be constructed: if a type cannot be named, the reference cannot be held. Because ownership contexts have been embedded in types, controlling aliasing reduces to controlling what objects have access to what ownership contexts.

Ownership contexts are accessible in a few ways. As the base case: every object can access its own ownership context using the overloaded **this** keyword; the **world** ownership context can be accessed using **world** keyword; and an object can access its owner's ownership context using the **owner** keyword. Next, ownership types allow an arbitrary number of extra ownership contexts to be passed to an object, as type parameters, with the restriction that all parameters are ancestors of the **owner** in the ownership tree.

The following code snippet shows an example of these concepts in the syntax of the Joline language [11]:

```
class Bar {}

class Foo<P1 outside owner> {
  this:Bar owned_by_me;
  owner:Bar owned_by_my_owner;
  owner:Foo<P1> same_type_as_me;
  this:Foo<owner> can_access_my_siblings;
}
```

Because every class must take an owner parameter, Joline makes the owner parameter implicit. Other ownership parameters are declared between angle brackets, like type parameters. Ownership parameters are bounded to be *outside* other parameters (meaning an ancestor in the ownership tree), with **owner** as the most general bound. When supplying the actual parameters to a class, the owner

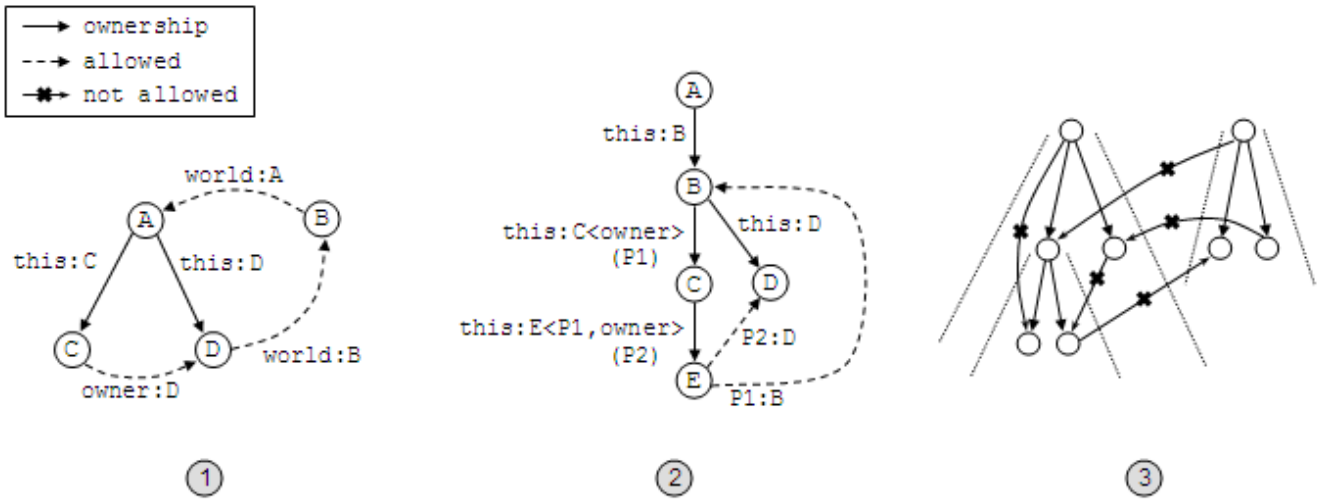

Figure 2. Examples of aliases allowed and not allowed by ownership types

is also distinguished from the other parameters by placing it before the type, separated by a colon.

Figure 2 illustrates the effect of ownership typing on the object graph. Diagram 1 shows the use of only **this**, **owner**, and **world**. Nodes represent objects and are labelled with the object's class. Arrows represent references between objects and are labeled with the type of the reference. Thus, the arrow labeled **this:C** indicates that the object of class **C** is owned by the object of class **A**.

The second diagram shows how additional ownership parameters can allow objects to access the ownership contexts of owners higher up in the ownership tree. For example, **E** can reference **D** because **E** has access to the ownership context of **D**'s owner, **B**. The identifier in parenthesis is the name of the formal parameter.

The third diagram shows references that are not allowed based on the ownership tree. Looking at the pattern of what references are and are not allowed, we can see one-way boundaries emerge on the object graph (drawn by the dotted lines). Visualizing these boundaries can help in understanding ownership types. A more formal statement is that ownership types guarantee the *owners-as-dominators* property on the object graph: an owner is a dominator on the path from **world** to all objects it transitively owns [2].

This forms the core of ownership types. On top of this, there are three additional extensions that need to be discussed. The first is the ability to parametrize a class by another class. OGJ allows type parameters and ownership parameters to be mixed compactly as follows:

```
class Box<Node extends Object<NodeO>> {
  Node held_in_box;
}
```

In this code, **NodeO** is the owner of **Node** and can be used to instantiate new classes. However, a subtle result of OGJ's treatment of ownership parameters and Java's type erasure semantics for generics is that **Node** represents a class that has already been instantiated with an owner. This means it is an error to try to give it a new type because:

```
class Outside<Inside extends Object<O>> {
  this:Inside mine; // wrong
}
```

really means (swapping the formal parameter **Inside** with the actual parameter **SomeType**):

```
class Outside<Inside extends Object<O>> {
  this:O:SomeType mine; // wrong: two owners
}
```

and supplying two owners is obviously wrong. What is needed is to pass an uninstantiated class that can be instantiated with arbitrary ownership parameters. This would be analogous to the "template template parameter" mechanisms in C++ and will be denoted in the parameter list by using the **class** keyword:

```
class Outside<class Inside> {
  this:Inside mine; // OK: Inside not already instantiated
}
```

Uninstantiated class parameters will be used extensively by the library types in the next section.

Another extension, which is also part of OGJ is *manifest ownership*. This allows a class to hard-code its owner by inheriting from a class instantiated with an owner:

```
class Foo extends world:Object { ... }
```

Written this way, **Foo** cannot be given an owner and will be the sibling of all **Foo**s in the ownership tree.

The last extension is owner polymorphic methods, which are part of Joline. This feature is one of several extensions which offer "principled violations of the ownership type system" (e.g., as described in [12]). Generally, such extensions are included to support common constructs such as iterators [13]. An owner polymorphic method lets the caller give the callee access to an ownership context for the duration of the call:

```
class Person {
  <You inside world> void lend(You:Gold yours) {
    You:Gold local_ref = yours;
    // mine = yours; (error)
  }
  this:Gold mine;
  // You:Gold stolen; (error)
}
```

This example shows how **You** is only available for the duration of the call, so references to the **You:Gold** cannot live past the call. This gives the concurrency library a tool to allow *temporary* aliasing between two objects dynamically determined to be in the same thread without the possibility that a reference will escape.

The height of the ownership tree described in this section is at most three. This might suggest a lighter-weight type system, like Universes [14], to achieve the same static guarantees. However, (1) the extensions used are based on an ownership type system, and (2) using this library in combination with similar libraries, as described in Section 5, involves nesting, which creates more complicated ownership trees requiring the full owners-as-dominators guarantee.

3.2 Typing

This section describes how the concurrency library can use ownership types as a tool to prevent data races, analogous to how a library can use **const** or accessibility modifiers to prevent clients from modifying returned references or accessing implementation details. The facility that ownership types add is statically enforced aliasing boundaries. By creating aliasing boundaries around threads and travelling objects, the concurrency library can guarantee to the library user: *if you can hold a reference to an object, it is safe to access it.*

The first step is to disallow client usage of **world**, which would allow allows client code to make and reference objects that are not local to any thread. The library types that are roots of the various ownership subtrees can then use the manifest ownership feature described in Section 3.1 to allow creation by library users without mentioning **world**. We can now revisit the parts of the ownership library that were introduced in Section 2. First, we consider the modified **AssemblyLine**:

```
class AssemblyLine<class GenericHost> extends world:Object {
  this:GenericHost host;
public:
  void send(Traveller<GenericHost::Guest>);
  void receive_returning();
}
```

AssemblyLine takes an uninstantiated **GenericHost** parameter and instantiates it with **this**. Without the **world** ownership context available, all objects created by **host** will necessarily be owned by the **AssemblyLine**. To guarantee that only travelling objects get moved between threads, **send()** only accepts the **Traveller** wrapper type, which is shown next:

```
class Traveller<class TravObjT> extends world:Object {
  this:TravObjT obj;
public:
  <O inside world> Traveller(O:TravObjT::InitArgs a) {
    obj = new<O> this:TravObjT(this, a);
  }
  <O inside world, class LocObjT>
  this:Tether<LocObjT> create_tether(O:LocObjT);
}
```

Traveller uses the same technique as **AssemblyLine** for owning a generic object. However, **TravObjT** cannot be default constructed like **GenericHost**, so **Traveller** takes in generic initialization data to pass to **TravObjT**'s constructor. Since **create_tether()** returns a **Tether** owned by **this**, only travelling objects can create tethers. To prevent direct construction, **Tether** has a private constructor:

```
class Tether<class PtrT> {
  // private constructor, only available to friend Traveller
  PtrT ptr;
public:
  bool alive();
  void request_access(owner:TetherUser<PtrT> p) {
    if (... same thread ...)
      p.access_granted(ptr);
    else
      ... error
  }
}
```

```
interface TetherUser<class PtrT> {
  <O inside world> void access_granted(O:PtrT);
}
```

Tether's main job is to guard access to the pointee. To do this, **Tether** requires that its users implement the **TetherUser** interface. Similar to the double virtual dispatch in the Visitor pattern, **access_granted()** gets called by **Tether** in response to calling **request_access()**. This approach does three things for **Tether**: first, it lets **Tether** dynamically guard access to the reference; second, the owner polymorphic method **access_granted()** allows **Tether** to prevent the given reference or any copies from outliving **access_granted()**; and third, **Tether** knows the duration of the reference's visibility and can thus prevent the travelling object from getting sent to another thread somewhere in the call stack.

As shown in the **Tether** pseudo-code, the library implementation ignores ownership types internally: **Tether** stores a plain reference to the object and calls **access_granted()** without any ownership context. This is similar in spirit to how, for example, a C++ **std::vector** presents a typed container interface to its users, but internally works with **malloc()**s, **void***s, and **memcpy()**s. As with **world**, this exemption should only exist for classes that are part of the library.

Lastly, client objects in the main thread need an owner. Without **world** and starting in the non-member **main()** function, however, there is no way to create objects. Following the same pattern as **AssemblyLine** and **Traveller**, **MainThread** allows the client to generically embed an object which will be owned by the **MainThread** object:

```
class MainThread<class ClientMain> extends world:Object {
  this:ClientMain host = new this:ClientMain;
public:
  int main() { return host.main(); }
}
```

With the library types covered, we can now consider what ownership types are needed for the user's code. The **Host** needs to modify its **arrived()** member function which gets called by **AssemblyLine** to reflect that it can only reference the arrived travelling object temporarily:

```
class Host {
  owner:Data data;
public:
  typedef Job Guest;
  <O inside world> void arrived(O:Job guest) {
    guest.do_work<owner>(data);
  }
}
```

The **data** member is a local object, so it is owned by the **AssemblyLine**. To pass a reference to the **Job**, **Host** needs to pass **owner** as well. As the travelling object, **Job** requires the most modifications:

```
class JobArgs {
  ...
  owner:Monster m;
}

class Job implements TetherUser<Monster> {
  owner:Tether<Monster> tether;
public:
  typedef JobArgs InitArgs;
  <O inside world> Job(Traveller t, O:JobArgs j) {
    tether = t.create_tether<O,Monster>(j.m);
  }
```

```
  <O inside world> void do_work(O:Data d);
  void welcome_back() {
    tether.request_access(this);
  }
  <O inside world> void access_granted(O:Monster m) {
    m.found_path(...);
  }
}
```

First, to support initialization in the generic **Traveller**, **Job** has to specify what data it needs with the **JobArgs** class and **InitArgs** associated type. **Job** also receives its owning **Traveller** as a constructor parameter, which it uses to create a tether. In **welcome_back()**, **Job** calls **request_access()**, passing itself to be the receiver of the **access_granted()** call. Instead of creating a **Job** directly, **Monster** now makes a **Traveller**:

```
class Monster {
  AssemblyLine<Host> ai_thread;
public:
  void think() {
    if (... I want to attack ...)
      ai_thread.send
        (new Traveller<Job>(new owner:JobArgs(this)));
  }
  void found_path(...);
}
```

Having the **Job** embedded in the **Traveller** prevents **Monster** from holding any references to the travelling object when it leaves.

In summary, the library requires all user objects to be owned by a library object. User objects that share the same owner are statically guaranteed to be in the same thread. Additionally, objects that are temporarily in the same thread can be allowed to reference each other in a controlled manner using owner polymorphic methods. A key part of this approach is that ownership types are not modified to include concepts of thread, local, travelling, and tethers. Rather, these concepts are in the library, which then uses ownership types as a tool for library design.

4. Embedding Ownership

Section 3 demonstrates how the primitives provided by the concurrency library of Section 2 could be checked if everything is written in an idealized language with ownership types. What is needed is a translation to this checkable form from Standard C++. We do not have such a translation implemented, however we outline what we believe is a promising approach to a minimal embedding in the language.

The first problem to address is how to attach ownership to references. In the simplest case, no annotation is needed at all. First, references to the library types that use manifest ownership (**AssemblyLine**, **Traveller**, and **MainThread**) do not need any ownership parameters. Next, when an owner is needed, **owner** may be used as a default. Defaulting has already been applied to Ownership Generic Java [15] to allow Generic Java programs to compile unmodified. For users of the concurrency library, code that does not deal with travelling objects will only refer to objects owned by the same thread. Thus, depending on how much code deals with concurrency, having **owner** be the default can eliminate much of the need for annotations.

When the default does not work, the programmer needs to make an annotation. There are many ways a programmer could make explicit the intent that a pointer or reference should represent ownership. The goal is to allow programmers and tools to verify that the ownership rules are obeyed. The most primitive approach is to use a special class of names for variables such as:

```
Foo *this_owned_a;
Foo *owner_owned_b;
```

where portions of identifiers are used as cues. Another approach is annotations in smart comments, which is the approach used by Universes [14]:

```
/** this: */Foo *a;
/** owner: */Foo *b;
```

However, the least intrusive interface to an analysis tool is a trivial template wrapper, such as:

```
this_owned_ptr<Foo> a;
owner_owned_ptr<Foo> b;
```

The templates are defined as any other template, using the standard syntax of C++. The type checker, however, can recognize the templates as an explicit ownership annotation. In addition to providing a solid handle for an analysis tool to work on, the wrappers can naturally introduce or remove operations on the wrapped type. The reason for using a technique that does not require language changes is that we eventually want to handle a large class of annotations and do not want to define our own set of dialects with their own compiler infrastructure. This is the SELL (Semantically Enhance Library Language) approach which we support with a simple tools infrastructure called "The Pivot" [16].

Considering in particular annotations needed for the concurrency library, the primary case is when using an owner polymorphic method:

```
class Job {
  <O inside world> void do_work(O:Data data) {
    // use data reference
  }
}
```

To annotate **data** we can write the following:

```
class Job {
  void do_work(caller_owned_ptr<Data> data) {
    // use data smart pointer
  }
};
```

Here, the presence of the **caller_owned_ptr** template wrapper indicates to the translation to both declare an owner polymorphic parameter and bind it to **data**.

Aside from annotating references with ownership, some of the constructs of the library had to be changed to accommodate ownership types. In particular: global variables and non-member functions need to be wrapped into a global object owned by a **MainThread**; **Tether**s are "dereferenced" indirectly through a double dispatch instead of using the more natural arrow operator; and inheriting from **TravellerBase** is changed to embedding in **Traveller**. For these special cases, a translation from C++ should be able to make simple patterned substitutions. For example, consider the following:

```
A *global = new A;
void foo(A *a) {}
int main() { foo(global); }
```

For type checking purposes, these globals can be collected into a single **Process** class that gets embedded in **MainThread**:

```
class Process {
  A *global = new A;
  void foo(A *a) {}
public:
  int main() { foo(global); }
};
```

```
int main() {
  MainThread<Process> mt;
  return mt.main();
}
```

With the default **owner** applied, the code can type check. A more involved example is converting uses of the arrow operator in **Tether** to double dispatch. Here, the translation involves hoisting the member function call, the arguments, and the return value into an automatically generated **TetherUser**. For example:

```
class Job {
  Tether<Monster> tether;
public:
  void welcome_back() {
    if (tether)
      tether->found_path(...);
  }
};
```

can be automatically translated into:

```
class AutoUser : public TetherUser<Monster> {
  ...
public:
  AutoUser(...);
  void access_granted(caller_owned<Monster> m) {
    m.found_path(...);
  }
};
class Job {
  Tether<Monster> tether;
public:
  void welcome_back() {
    if (tether)
      tether.request_access(AutoVisitor(...));
  }
};
```

With these and related transformations, the syntactic burden over normal use of the library can be reduced while internally generating the fully ownership-annotated source for checking.

5. Discussion

The presentation so far has been concerned with describing our experience with a single library on a single project. This section branches out to consider a wider range of features and applications of this kernel experience.

5.1 Variations on Tethers

The **Tether** construct presented in this paper was motivated by the specific needs of a project, but other variations on the same approach make sense for different situations. The essential ideas are: (1) regardless of aliasing boundaries, objects need to be able to point to objects in other threads, and (2) these pointers can have different operations in place of the standard "dereference". We present two further examples here.

An opposite approach to calling **AssemblyLine::send()** is for a local object to use a **Tether** to "pull" a travelling object into the same thread. The pull operation waits until the target object is not in use in its current thread and transfers it to the caller's thread.

```
class Worker {
  PullTether<RenderPipeline> rpipe;
  void render_data(...) {
    // prepare data for rendering
    // expensive computation...
    PulledObject<RenderPipeline> po = rpipe.pull();
    po->render(...);
  }
};
```

We can see that these semantics are analogous to that of a traditional lock which protects the object getting pulled. However, without any additional work on the part of the user, the runtime system can make optimizations over plain locks. First, by keeping track of the tethers to an object, the runtime can tell which threads can possibly request a lock at the same time. With this knowledge, the runtime can use cheaper locks when, for example, it knows that all contending threads are assigned to the same physical processor. Conversely, in a non-uniform memory architecture, the runtime system could look at the tethers that exist between objects and place threads which have many tethers between them "closer" together, with respect to the machine topology.

Another variation is to treat a tether as a homing device for the object to which it points. Instead of pulling a distant object close, tether could be augmented to provide a "take me to this object" operation which allows a travelling object to go to the thread that owns the pointee:

```
class UpdateCourier {
  Update update;
public:
  void update_data(HomingTether<Data> d) {
    d.go_to_thread(*this);
  }
  void arrived_at_thread(Data &d) {
    d.apply(update);
  }
};
```

In this example, a control thread updates data structure that are local to different processing threads by sending courier objects to the threads with the update. Courier objects are given **HomingTether**s to indicate which data set needs to receive the update. Finally, **arrived_at_thread()** is called by **HomingTether** when the transfer is complete.

5.2 Variations on Libraries

In this section we identify the design and typing of the concurrency library as an instance of a more general pattern of library design. The pattern is defined by: (1) providing library primitives whose semantics imply aliasing boundaries, and (2) providing the user of the library semantically-modified pointers to refer across these boundaries. We now consider two other examples, how their primitives imply aliasing boundaries, and how users can refer to objects across these boundaries.

5.2.1 Memory Protection

Fine grained memory protection has been used for security, fault isolation, and efficient IPC since early capability-based architectures [17] and continues to be researched. Recent work includes Mondriaan Memory Protection (MMP) which has been applied to the Linux kernel [18]. The idea is to associate *protection domains* with allocated memory regions and threads. Threads are then prevented from accessing memory outside their current protection domain. This approach helps find errors that might have gone undetected and catches errant program behavior closer to the source.

A straightforward API for a memory protection library would provide functions for: allocating and deallocating opaque protection domain handles; adding and removing memory regions to and from domains; and changing the domain of the currently executing thread. These API calls could be abstracted by an object-oriented library in the same manner that the concurrency library in Section 2 abstracted low level locking and thread operations. The aliasing boundaries in this case would align with protection domains and a library pointer type would be provided to point to objects in other protection domains. The library could then either offer travelling mechanisms similar to the concurrency library or simply provide

a dereference operation. In addition to allowing a flat partitioning of memory, systems like MMP allow a region of memory to be in more than one protection domain. This lets the user create a nesting structure of permissions which directly corresponds to the owners-as-dominators property enforced by ownership types.

5.2.2 Resource Accounting

A good operating system will release all resources requested by a process when the process exits. This requires the system to record which resources have been allocated by the process. Thus, a simple way to do "garbage collection", not only for memory but all OS resources, is to fork child processes to handle work items and then exit, automatically freeing the resources used to process the work item. This approach has several performance disadvantages and consequently developers usually need to use multiple threads and careful resource management instead.

The utility of a process, with respect to resource management, is that it provides a single collection point for resources. To achieve the same effect at a finer granularity, we can introduce "resource domains". Each resource domain owns a set of objects and keeps track of all allocation requests made by objects it owns. One challenge for the library is to keep track of the current resource domain as execution passes between objects owned by different resource domains. By aligning aliasing boundaries with resource domains, the library user would be required to use a library mechanism when pointing to objects in other resource domains. By controlling access to objects in other resource domains, the library can keep track of changes:

```
class EnemyAI : ResourceDomainVisitor<EnemyGraphics> {
  CrossDomainPtr<EnemyGraphics> ptr;
public:
  void think() {
    if (... decide to hold a fireball ...)
      ptr.access_resource_domain(*this);
  }
  void in_resource_domain(EnemyGraphics &g) {
    // allocate Fireball in graphics resource domain
    g.shoot(new Fireball);
  }
};
```

In this example, the AI component of an enemy creates a **Fireball** for the graphics component to show. The two objects are in different resource domains, so the **EnemyAI** needs to use the library-supplied pointer type **CrossDomainPtr**. To access the object, the same double virtual dispatch technique used by **Tether** in Section 3 is used. This allows the library to change domains for the duration of **in_resource_domain()** so that **Fireball** is allocated in the graphics resource domain.

Hierarchical resource management is normally done in C++ using constructors and destructors following the Resource Acquisition Is Initialization idiom [19]. On the opposite end of the resource management spectrum, garbage collection tries to hide when resources are released and does not associate an owner. The approach presented in this section is therefore somewhere in between: resources have owners and deterministic bounds on their allocation, but these bounds are more like catch-alls than proper manual resource management. Thus, allocation domains can be seen as a fine-grained way to handle leaks or a way to recover resources when an error has left a portion of the system in an undefined state.

5.2.3 Summary

In the examples above, the library provides primitives that organize objects in the program hierarchically. To fully utilize this library design, however, several libraries need to be able to coexist in the same ownership tree in the same program. For example, consider a modern web browser. Concurrency boundaries can be associated with different browsing windows, security boundaries with the scripting interpreters, memory protection boundaries with less-than-stable modules, and resource accounting boundaries where leaks are difficult to avoid. This implies a heterogeneous nesting of boundaries which we have not considered thus far. For the same reason it is necessary to cross homogeneous boundaries, it will be necessary to compose each library's semantically-modified pointers to cross multiple heterogeneous boundaries. This ventures far from the experience and example focused on by this paper but we feel it points to an exciting use of ownership types as a tool for future library design.

6. Related Work

Since the widespread recognition of the problems of aliasing in object-oriented programming, and the need for local reasoning, more than a decade ago [1], many type systems have emerged to address the problems. The approaches vary from completely outlawing aliasing using variants of linear types [20, 21], to cutting the object graph into fully encapsulated partitions [22, 23], to enforcing an owners-as-dominators property on the object graph using ownership types [2], to even more flexible and/or less intrusive type systems with less guarantees [24–26]. Of these approaches, ownership types have emerged as a promising compromise and many different aspects of the type system have been researched [3, 27, 28]. Boyapati *et al.* have used and extended ownership types to guarantee the absence of data races and deadlocks [4], statically safe region-based memory management [5], and safe lazy upgrades to persistent object stores [6].

The work most similar to ours is SafeJava [4], which also uses ownership types. More recent work to statically ensure the absence of data-races has been done by Jacobs *et al.* using automatically verified annotations in the Spec# compiler [29, 30]. The main difference between our approach and these two is the basis for concurrency: in our model, nothing is shared and objects travel between threads; in the other two, there are shared objects which are owned by **world** and synchronized with locks. SafeJava does allow unique types to be passed between threads via a synchronized global shared variable, but this places aliasing constraints on the unique object which would not allow constructs like tethers. Another difference is how the data-race freedom guarantees are made. These approaches use concurrency constructs built into the language and build concurrency guarantees into the type system. In the approach we outline, the library both provides the concurrency primitives and uses a generic ownership type system to make guarantees about use of the library.

7. Conclusion and Future Work

In this paper we presented a simple library for concurrency, successfully used in a large student project, and demonstrated how ownership types could be used to statically check that client code respect the aliasing boundaries imposed by the library. To provide flexible support for objects travelling between threads while carrying aliases to thread local objects, we combine owner polymorphic methods with dynamic checks performed by the library to guarantee the absence of data races. Finally, we present an approach to embed the necessary ownership annotations in C++ and to use an extended type checker to enforce the rules on top of the language.

We also found the strategy used to support the concurrency library was also found to apply to a family of related libraries including memory protection and resource accounting. One direction for future work is to examine existing programs that exhibit task-level parallelism, like the videogame example in this paper. By looking at more and larger programs, we can further develop both the concur-

rency model and typing approach introduced here to address more usage scenarios.

Acknowledgements: We are grateful to the reviewers for their feedback, members of the program committee for their patience, and Alex Potanin for his help.

References

[1] John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux, and Richard Holt. The Geneva convention on the treatment of object aliasing. *SIGPLAN OOPS Mess.*, 3(2):11–16, 1992.

[2] David Gerard Clarke. *Object ownership and containment*. PhD thesis, University of New South Wales, 2002.

[3] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Generic ownership for generic Java. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 311–324, New York, NY, USA, 2006. ACM Press.

[4] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 211–230, New York, NY, USA, 2002. ACM Press.

[5] Chandrasekhar Boyapati, Alexandru Salcianu, Jr. William Beebee, and Martin Rinard. Ownership types for safe region-based memory management in real-time Java. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 324–337, New York, NY, USA, 2003. ACM Press.

[6] Chandrasekhar Boyapati, Barbara Liskov, Liuba Shrira, Chuang-Hue Moh, and Steven Richman. Lazy modular upgrades in persistent object stores. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 403–417, New York, NY, USA, 2003. ACM Press.

[7] K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516. Springer-Verlag, 2004.

[8] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. *Sci. Comput. Program.*, 62(3):253–286, 2006.

[9] Texas Aggie Game Developers: `http://tagd.cs.tamu.edu`.

[10] C++ Boost Library Collection, 2007. Boost Smart Pointers: `http://www.boost.org/libs/smart_ptr/smart_ptr.htm`.

[11] Tobias Wrigstad. *Ownership-Based Alias Management*. PhD thesis, Stockholm University, 2006.

[12] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *ACM Symposium on Principles of Programming Languages (POPL)*, New Orleans, Louisiana, January 2003.

[13] James Noble. Iterators and encapsulation. In *TOOLS '00: Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 33)*, page 431, Washington, DC, USA, 2000. IEEE Computer Society.

[14] Werner Dietl and Peter Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.

[15] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Defaulting generic Java to ownership. In *In Proceedings of the Workshop on Formal Techniques for Java-like Programs in European Conference on Object-Oriented Programming*, Oslo, Norway, 2004. Springer-Verlag.

[16] Bjarne Stroustrup. A rationale for semantically enhanced library languages. In *Proceedings of the First International Workshop on Library-Centric Software Design (LCSD '05)*, 2006. As technical report 06-12 of Rensselaer Polytechnic Institute, Computer Science Department.

[17] R. M. Needham and R. D.H. Walker. The Cambridge CAP computer and its protection system. In *SOSP '77: Proceedings of the sixth ACM symposium on Operating systems principles*, pages 1–10, New York, NY, USA, 1977. ACM Press.

[18] Emmett Witchel, Junghwan Rhee, and Krste Asanovic. Mondrix: memory isolation for Linux using Mondriaan memory protection. *SIGOPS Oper. Syst. Rev.*, 39(5):31–44, 2005.

[19] Bjarne Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[20] Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, pages 347–359. North Holland, 1990.

[21] Dave Clarke and Tobias Wrigstad. External uniqueness is unique enough. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 176–200, 2003.

[22] John Hogg. Islands: aliasing protection in object-oriented languages. In *OOPSLA '91: Conference proceedings on Object-oriented programming systems, languages, and applications*, pages 271–285, New York, NY, USA, 1991. ACM Press.

[23] Paulo Sérgio Almeida. Balloon types: Controlling sharing of state in data types. *Lecture Notes in Computer Science*, 1241:32–59, 1997.

[24] Jan Vitek and Boris Bokowski. Confined types. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 82–96, New York, NY, USA, 1999. ACM Press.

[25] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Featherweight generic confinement. *J. Funct. Program.*, 16(6):793–811, 2006.

[26] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. *SIGPLAN Not.*, 37(11):311–330, 2002.

[27] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. *SIGPLAN Not.*, 37(11):292–310, 2002.

[28] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Safe runtime downcasts with ownership types. Technical Report TR-853, MIT Laboratory for Computer Science, June 2002.

[29] Bart Jacobs, Jan Smans, Frank Piessens, and Wolfram Schulte. A statically verifiable programming model for concurrent object-oriented programs. In *8th International Conference on Formal Engineering Methods*, pages 420–439, 2006.

[30] Bart Jacobs, Frank Piessens, K. Rustan M. Leino, and Wolfram Schulte. Safe concurrency for aggregate objects with invariants. In *SEFM '05: Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*, pages 137–147, Washington, DC, USA, 2005. IEEE Computer Society.