

A Polymorphic Future and First-class Function Type for Concurrent Object-Oriented Programming*

R. Greg Lavender

Dennis G. Kafura

ISODE Consortium
8920 Business Park Dr.
Austin, Texas 78759
g.lavender@isode.com

Department of Computer Science
562 McBryde Hall, Virginia Tech
Blacksburg, Virginia 24061
kafura@cs.vt.edu

To be submitted to the Journal of Object-Oriented Systems

Abstract

A “future” is a typed synchronization mechanism used to represent a value that will come into existence at some point in time after the creation of the future. In this paper, a *polymorphic future type* is introduced that differs from previous mechanisms in that it provides a first-class, extensible facility for expressing synchronized access to the typed result value of a concurrent computation. The proposed mechanism requires *write-once/read-many* synchronization, which is implemented in terms of common thread synchronization primitives, thereby ensuring a high-degree of efficiency. The type definition and type specialization features of C++ are used to illustrate how one would implement a polymorphic future type in a strongly typed object-oriented language supporting polymorphic type definitions.

1 Introduction

Applications increasingly employ multiple threads of control due to the utility of the thread mechanism and its widening availability. The utility of threads is apparent in any application that seeks to exploit the performance advantages of multiprocessor systems or that must manage a variety of asynchronous events, such as those occurring in windowing systems or in distributed systems using remote procedure invocations. Threads are available as intrinsic facilities in modern operating systems, such as Mach, OSF/1, OS/2, and NT, and through user-level run-time libraries in older, process-oriented systems such as UNIX.

Object-oriented languages can facilitate programming with threads by providing class definitions for two key abstractions: a thread abstraction and a future abstraction. The need for a thread abstraction is obvious and is the usual focus of operating systems and run-time packages. Less obvious, but equally critical, is the need to manage the value returned from an asynchronously invoked operation. Coroutines, message send operations, remote procedures, and local parallel procedures are examples of operations that may be asynchronously invoked. The term “future” has been used to identify the mechanisms surrounding the generation of and access to such a returned value. The word future is used to suggest that the returned value, if not currently available, will be available at some time in the future. Because of the time element involved, the management of the returned value involves a concern for synchronization in addition to the usual concerns for type checking.

The challenge taken up in this paper is the definition of a type abstraction that is syntactically convenient, expressive, and which incurs minimal execution overhead. Since the implementation makes very

*This work was supported in part by National Science Foundation grant CCR-9104013.

modest assumptions about the underlying synchronization mechanisms, we believe that the polymorphic future type can be used with a wide range of existing threads facilities such as: the C threads library provided with Mach [5], the Lightweight Process (LWP) library provided with SunOS or Solaris [23, 24], the POSIX threads library [12, 19], and the PRESTO threads package [1, 2].

In Section 2, the structure of a general future mechanism is described, the condition synchronization issues are discussed, and an implementation in C++ is then presented. In Section 3, the new future mechanism is used in the construction of a new concurrent programming abstraction, called a *lambda type*, which is used to define functions as first-class objects. The polymorphic future mechanism is contrasted with previous future mechanisms in Section 4.

2 The Structure of a Future Mechanism

A principle feature of a concurrent object system is the ability to construct user-defined objects that contain operations possessing an independent thread of control when invoked. The set of instance variables encapsulated by the object represent a shared state among the concurrent operations defined by the object. Figure 1 contrasts the difference between synchronous and asynchronous interaction of a client object with a server object.

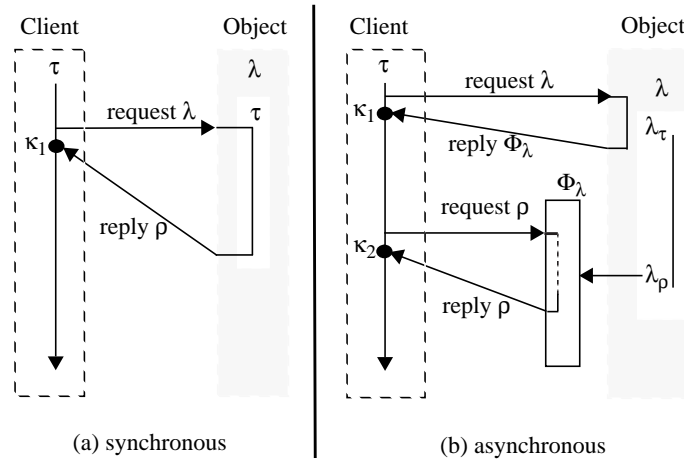


Figure 1: Method Execution Model.

Part (a) depicts the traditional synchronous, stack-based, request/reply interaction between a client and a method defined as part of an object. The client synchronously invokes the method denoted by λ with the appropriately typed argument values. The request operation establishes an implicit *continuation* point, κ_1 , at which control will resume when the method returns. The called method is executed using the *thread*, τ , of the caller. Upon completion of the λ method, control resumes at κ_1 with the result value denoted by ρ . The advantage of this interaction model is that it is simple, efficient, and amenable to common optimization techniques, such as inlining. The disadvantage is that the nested, stack-based model is inherently sequential.

Part (b) depicts an asynchronous interaction between a client and an object method. The invocation establishes an implicit continuation point, κ_1 , for the client. The method invocation mechanism uses the client's execution thread, τ , to create a new thread environment, λ_τ , within which the λ method will execute. The invocation mechanism then resumes at the continuation point κ_1 of the client. The client thread τ and the λ_τ thread may then execute concurrently.

```

template<class T> class Future : private Ref< Capsule<T> > {
public:

    inline Future() : Ref< Capsule<T> > (new Capsule<T>()) {}
    inline Future(const Future<T>& f) : Ref< Capsule<T> >(f) {}

    inline void operator=(const T& r) { **ref = r; }

    inline operator T() { return **ref; }
    inline T operator*() { return **ref; }

    inline int ready() { return (*this)->readable(); }
};

```

Figure 2: Polymorphic Future Type.

The result of an asynchronous invocation is a value denoted by an object Φ_λ . When a client resumes execution at κ_1 , it has a capability to manipulate the Φ_λ object as a though it were the result value, ρ , of the λ operation. The Φ_λ object represents a *future*, from which a client can request the future result, λ_ρ , of the execution of the corresponding λ operation. At a later point in time, the original client thread, or any other thread that has access to the future, may request the result of the λ operation from the Φ_λ object, depicted as a request for ρ . If the result from the concurrently executing λ method has not yet been bound to the future, then the client thread blocks in the Φ_λ object. When the result is computed and the thread τ_λ terminates, the client’s continuation point κ_2 is resumed with the result value ρ . In effect, the future is “converted” to a value, either implicitly or explicitly, when the result of the corresponding computation is produced. Subsequently, the Φ_λ object and the thread τ_λ are subject to garbage collection.

A mechanism that implements a future object as just described requires the following characteristics:

- polymorphic type definition
- write-once, read-many condition synchronization
- garbage collection

Figure 2 illustrates the definition of a polymorphic future type expressed as a template class parameterized by the generic type parameter T. A polymorphic future type is referred to as a “future of type T” and denoted by the type `Future<T>`. In the common case, the type parameter T defines the type of the return value of a procedure. A concrete future type is an instance of `Future<T>`; for example, a `Future<int>` denotes a future of type integer. To illustrate the use of a polymorphic future type, consider the following type signature for a `connect` operation that performs an asynchronous network connection request returning an integer result once the connection is established, aborts, or is rejected:

```

Future<int> connect(...);    /* asynchronous network connect */

```

A natural consequence of the representation of a future as a polymorphic type is that an instance of a specific `Future<T>` is implicitly convertible to a instance of type T. That is, a `Future<T>` instance can be used in place of an instance of type T—conversion will occur implicitly. The semantics of future type conversion are that a value of type T is produced if the value has been computed; otherwise, the type conversion to type T implicitly blocks awaiting the result. This semantics implies that synchronous invocation can be accomplished simply by requesting type conversion at the site of invocation; for example:

```

int result = connect(...);    /* implicitly blocks */

```

The `connect` operation returns a `Future<int>` object that is implicitly converted to an `int` since `result` is an integer type. The caller blocks at the expression continuation of the assignment statement awaiting the result value of the `connect` operation. Non-blocking, asynchronous behavior is achieved by explicitly using a `Future<T>` instance and later requesting the result value by implicit type conversion, as follows:

```
Future<int> future = connect(...);    /* async invocation */
:
int result = future;                 /* might block */
```

An inherent drawback to the use of futures in a statically typed language like C++ is that arguments to procedures are passed by value. Eager evaluation of procedure arguments forces the conversion of a future type to its result value, possibly blocking the procedure invocation awaiting the future value for an argument. Applicative-order evaluation implies that it is not possible within the C++ type system to pass a `Future<T>` instance to a procedure expecting a type `T` argument value without causing the evaluation of the future. If eager evaluation is not desired, it is necessary to explicitly specify an argument of type `Future<T>` in the type signature of the operation, thereby effecting call-by-need.

2.1 Future Synchronization

A future may be viewed a synchronization object that is the site of a rendezvous between a thread computing the value represented by a future, and the client thread(s) that will eventually request the value. The act of requesting that a future type convert itself from a future of type `T` to a value of type `T` is where synchronization comes into play. The potential for concurrent access to a `Future<T>` object dictates a requirement for synchronization control to guarantee that reader threads block on the future until the thread writing the future produces the result. Reader threads themselves may execute concurrently within a future. The condition synchronization requirements of a future are similar to traditional reader-writer condition synchronization, which is solvable using mutual exclusion and condition variables. A *capsule* is an object that is used by a future to encapsulate the eventual result value and provide synchronized access. The reason for instantiating a capsule object independent of a future object is primarily for efficiency. The synchronization employed by a future capsule requires mutual exclusion and condition synchronization. A novel approach is to define a capsule as part of a synchronization type hierarchy for monitors that uses inheritance to allow monitor specialization. Using inheritance, a capsule may be implemented as a subtype of a generalized reader-writer synchronizer (monitor) whose behavior is specialized, through inheritance, to provide the write-once, read-many synchronization semantics required by a future.

Figure 3 illustrates the relationship between a `Future<T>` object, a `Capsule<T>` object, and a result object of type `T`. As stated previously, the construction of a `Future<T>` object results in the implicit construction of a corresponding `Capsule<T>` object. The solid link from the `Future<T>` object to the `Capsule<T>` object denotes that the binding between the two objects is established at the time a `Future<T>` object is constructed. The `Future<T>` class is purposely separated from the `Capsule<T>` class so that a `Future<T>` instance may be used efficiently as either a return value or argument to a procedure. The dashed link between the `Capsule<T>` object and the object of type `T` is established when a value is computed and bound to the future capsule.

The shaded box surrounding the `Capsule<T>` object in Figure 3 depicts the composite object created by a three level inheritance structure. The inheritance hierarchy is *private*, meaning that only a `Capsule<T>` object is visible to a `Future<T>` instance. As depicted, the `Capsule<T>` class inherits privately from the `RWSync` class, which in turn inherits from the `Mutex` class to enable mutual exclusion. The `RWSync` class defines a monitor that implements a producer-consumer synchronization discipline extended to provider concurrent consuming broadcasting to all blocked consumers. The actual mutual exclusion mechanism is system dependent. The efficiency of `Capsule<T>` objects is limited by the

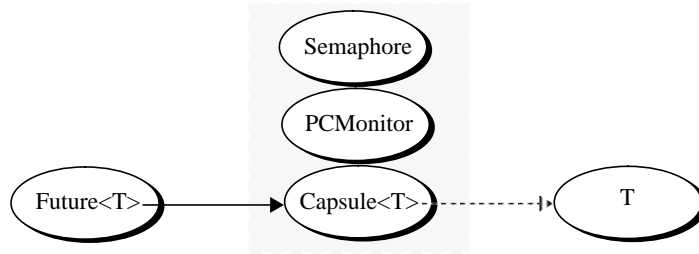


Figure 3: Relationship among `Future<T>`, `Capsule<T>`, and value `T` objects.

underlying system mechanism.¹

At some point in time, a `Capsule<T>` instance will bind to a dynamically created instance of type `T`, denoted by the dashed arrow to the object labeled `T`. The object of type `T` represents the eventual value of a future. Figure 4 illustrates the template class that defines the polymorphic capsule type `Capsule<T>`. The generic type parameter `T` corresponds to the same type parameter of a `Future<T>` since instances of `Capsule<T>` are instantiated and bound to a specific `Future<T>` instance by the `Future<T>` constructor.

A concurrent producer thread binds a value of type `T` to a `Future<T>` instance by assignment, and one or more concurrent consumer threads access that value by using either the type conversion operator, the dereference operator, or the member access operator described in the last section. A `Future<T>` instance must use a synchronization control mechanism to block any consumer threads that attempt to access a future object before the corresponding producer thread has bound a value to the future. Once the value is bound, all of the consumer threads can then access the future result value concurrently since a `broadcast` signal is delivered to the consumer condition queue.

2.1.1 Construction/Destruction

Construction and destruction of `Capsule<T>` types is only permitted by objects of type `Future<T>`. Placement of the constructor and destructor operations within the context of a `private` clause in the class definition, and declaration of the `Future<T>` class as a `friend` class, guarantees controlled construction and destruction of capsule objects.

Construction of a `Capsule<T>` instance occurs as the result of the construction of a `Future<T>` object. The default constructor simply initializes the variable `result` to a null value and initialized the reference count `refs` to 1.

Destruction of a `Capsule<T>` type is caused by the destruction of the `Future<T>` object to which it is bound. Destruction simply deletes the type `T` object bound to `result`, if any. If no result is bound and consumer threads are blocked on the consumer condition queue awaiting the result, the default action is to terminate all blocked threads.

2.1.2 Condition Synchronization Predicates

Inheriting synchronization constraints in concurrent object-oriented programming languages results in a conflict that arises when attempting to reuse methods that internalize condition synchronization constraints. As previously demonstrated, the conflict arises because the synchronization conditions are too tightly bound to the monitor operations. The solution outlined previously implements condition synchronization constraints as “virtual” synchronization predicates that are defined independently of

¹Recent work by Bershad [3] has shown that mutual exclusion in support of multi-threading on uniprocessor RISC architectures can be made highly efficient even in the absence of explicit hardware support.

```

template<class T> class Capsule : private RWSync {
private:

    T*   value;           /* generic value binding */
5

public:

    inline Capsule()      { value = 0; }
    inline ~Capsule()     { if (value) delete value; }
10

    inline void operator=(const T& r) {
        WriterMutex mutex(this);
        value = new T();
    }
15

    inline operator T() {
        ReaderMutex mutex(this);
        return *value;
    }
20

    /* overloading of inherited condition synchronization predicates */

    int readable()        { return RWSync::readable() && value != 0; }
    int writeable()      { return RWSync::writeable() && value != 0; }
25
};

```

Figure 4: Polymorphic Capsule Type.

monitor operations. The monitor operations utilize virtual synchronization predicates and the binding between the monitor operations and the predicates is thus relaxed. Defining synchronization conditions as virtual predicates allows the conditions to be easily redefined by derived classes, thereby facilitating the reuse of monitor operations defined by a base monitor class.

The two synchronization predicates, `readable` and `writeable`, are virtual methods inherited by the `Capsule<T>` class from the `RWSync` class. The `Capsule<T>` class provides an implementation for each predicate that is specific to the synchronization requirements of futures. The `readable` predicate is used by `RWSync` operations to enforce condition synchronization on reader threads; likewise, the `writeable` predicate is used to enforce condition synchronization on writer threads, of which there should only be one since the desired semantics are write-once. The `consumable` synchronization predicate is defined to be `TRUE` only if an object of type `T` has been bound to `result` (i.e., `result` is not `nil`). An object of type `T` is bound to `result` by a producer thread executing a `Future<T>` assignment operator. The `producible` predicate is defined to be the constant `TRUE` since it is always the case that a producer thread may bind an object type `T` to `result`, subject only to acquisition of mutual exclusion by the `Future<T>` assignment operator. Hence, by overloading the inherited condition synchronization predicates with future specific conditions, the behavior of the `RWSync` is specialized to meet the particular synchronization needs of arbitrary future types.

2.2 Future Methods

A key issue in the implementation of the `Future<T>` type is that the methods should be easily redefined through inheritance and the implementation should incur minimal execution overhead. Table 1 summarizes the method attributes of the `Future<T>` type.

The following observations are made concerning the methods defined in the `Future<T>` declaration depicted in Figure 2:

- all operators are defined as inline methods.

Table 1: Summary of Polymorphic Future Methods Attributes.

Methods	Attributes	Description
<code>c'tor</code>	public, inline, static	default constructor
<code>c'tor&</code>	public, inline, static	“copy” constructor
<code>create</code>	public, inline, static	used in place of private <code>new</code>
<code>d'tor</code>	public, inline	default destructor
<code>operator new</code>	private, inline, static	dynamic allocation prohibited
<code>operator delete</code>	private, inline, static	dynamic deallocation prohibited
<code>operator &</code>	private, inline	“address of” operator prohibited
<code>operator =</code>	public, inline, mutex	overloaded assignment operator
<code>operator T</code>	public, inline, mutex	type conversion operator
<code>operator *</code>	public, inline, mutex	overloaded dereferencing operator
<code>operator -></code>	public, inline, mutex	overloaded member access operator
<code>operator ==</code>	public, inline, const	overloaded equality relation
<code>operator !=</code>	public, inline, const	overloaded inequality relation
<code>exists</code>	public, inline, const	existence predicate

- all non-constant operator methods employ synchronization control to ensure consistent access to the value held by the future.

Method inlining results in faster code only if the compiler also performs efficient machine register allocation when replacing a procedure call with inlined code. Davidson and Holler [6] offer convincing empirical evidence that inlining improves performance more often than not, particularly for small procedures like those defined by the `Future<T>` template class. By declaring all `Future<T>` methods inline, an optimizing compiler can often reduce the run-time overhead associated with invoking `Future<T>` operations; thus, operations on future objects are generally more efficient than normal procedure calls, at the cost of increased code size at the call site.²

2.2.1 Construction/Destruction

As depicted in Figure 2, a `Future<T>` object consists of a single data item denoted by the *protected* instance variable `future`, which is a pointer to an object of type `Capsule<T>`. A `Future<T>` object is typically created by a procedure that returns the future as a result value of an asynchronous procedure invocation. The size of a `Future<T>` object is the amount of storage required to hold a pointer, which is typically no greater than four octets. Hence, a `Future<T>` object may be efficiently returned by a procedure through a register and passed by value as a procedure argument, requiring only one slot in the call frame. The lifetime, or extent, of a future object once created, is subject only to the number of threads that maintain a reference to the future. In the simple case, there are two references to a future:

- one for the producer thread that will eventually bind a value to the future, and
- one for the consumer thread that will eventually claim the value.

In both cases, the producer and consumer contexts dictate the lifetime of the future. When a future is no longer “live”, due to the fact that all contexts in which it is referenced no longer exist, it is deleted. Ideally, an automatic garbage collector would maintain such information and automatically delete future objects when they are no longer referenced. However, C++ does not currently provide automatic garbage collection of objects.³ As an alternative, a simple and efficient reference counting scheme based on the

²The C++ front-end to the GNU gcc 2.x compiler [21] performs efficient method inlining and generates optimized native machine code on a wide range of architectures.

³A future version of the GNU C++ compiler will employ some support for a generation-scavenging garbage collector, based on work by Moss et al. [11].

controlled construction and destruction of `Future<T>` objects is adopted. This scheme is based on the ability to restrict `Future<T>` objects so that it is not possible to obtain a pointer to a future object. Furthermore, each time a `Future<T>` object is constructed or destructed (either explicitly or implicitly as a result of scope rules) the reference count is updated properly.⁴

Constructors: A `Future<T>` object may be constructed either by a declaration of a `Future<T>` instance or by calling the `create` method. The `create` method is used in place of the `new` operator for dynamically creating future objects. The need for reference counting dictates that pointers to `Future<T>` objects must be prohibited; hence, the `new` operator, the `delete` operator, and the “&” operator for taking the address of an object, are declared as private methods to prohibit public use. Note that it is still possible to define a `Future<T>` reference type, denoted by the type `Future<T>&`. C++ reference types present a problem. References are like pointers, but are more restricted. A common usage is in the definition of a *copy constructor* used to “clone” objects. For example, the `Future<T>` copy constructor is defined by the following type signature:

```
Future(Future& f);
```

A copy constructor requires a reference type parameter; hence, unlike `Future<T>` pointer types, it is not acceptable to prohibit `Future<T>` reference types. Detecting when a future object is referenced by a reference type is problematic since the future reference counting scheme is based on the orderly construction and destruction of future objects. Assignment of a future object to a future reference type is not subject to the same rules. In principle, the reference count could be updated by defining an overloading for the `Future<T>&` type conversion operator in the `Future<T>` class definition. However, this would only allow the detection of when a reference type was applied to a future object; there is no general mechanism for detecting when the same reference is dropped since reference types do not have destructors. As long as a reference type is only used as an argument to a procedure, and the scope of the future object bound to the reference exists longer than the scope of the procedure, then object reference types may be safely used. No other use is deemed safe since it is potentially the case that the future object being referenced might be deleted by some other thread if the reference count reaches zero.

The default construction of a `Future<T>` type results in the initialization of the `future` instance variable to a null value. A `Capsule<T>` object is only instantiated when the future object is actually used, by invoking one of the `Future<T>` public operations. The decision to delaying the binding of a `Capsule<T>` object is an optimization to eliminate unnecessary overhead when constructing temporary future objects. The binding of a `Capsule<T>` object occurs immediately on invocation of the copy constructor. The copy constructor invokes the `clone` operation, causing the binding of a `Capsule<T>` object to the future object being cloned, if one is not already bound. The newly constructed future object is then bound to the *same* `Capsule<T>` object; that is, cloning is used to make future objects share the same `Capsule<T>` object.

Destructor: The destruction operation `~Future` decrements the `future` reference count and deletes the bound `Capsule<T>` object if the reference count is zero. The deletion of the `Capsule<T>` instance bound to `future` implies the destruction of the encapsulated type `T` object, if any.

2.2.2 Overloaded Operator Methods

The body of each operator method contains a local declaration of a `mutex` object that enforces either `Consumer` or `Producer` mutual exclusion. A `mutex` object of type `Consumer` enforces consumer synchronization while an object of type `Producer` enforces producer synchronization. Upon entry to any

⁴Reference counting schemes have limitations with respect to reference cycles; hence, situations in which a future references another future either directly or indirectly should be avoided. When used in a normal manner as the result of an asynchronous procedure invocation, cycles will not occur.

operator method, a `mutex` object of the appropriate type is instantiated for the `Capsule<T>` instance bound to `future`. For example, a method that returns the value of a future defines:

```
Consumer mutex(future);
```

Construction of the `mutex` object occurs automatically on scope entry and corresponds to Dijkstra's P operation combined with the appropriate condition synchronization predicates that determine whether or not to delay on a condition queue. The construction of a `mutex` object blocks if the synchronization condition for the specified type is not met. Implicit destruction occurs on scope exit and corresponds to the V operation combined with signal-and-continue semantics for the appropriate condition queue. For example, the destructor for a `Producer` instance releases mutual exclusion and all waiting consumer threads that they may continue by broadcasting to the consumer condition queue.

A semantically satisfying aspect of defining mutual exclusion and condition synchronization as a declaration of a `Mutex<T>` subtype whose extent is subject to normal procedure scope rules is that the synchronization protocol is correctly followed by all operations.

Assignment: The doubly overloaded assignment operator `operator=` is used to assign either a value of type `T` to a future object or another future. Since assignment updates the future, a `Producer` object is constructed upon entering scope to effect mutual exclusion. Once mutual exclusion is acquired, a new object of type `T` is constructed and assigned to the `result` variable of the `Capsule<T>` object referenced by the `future` variable.

A requirement of all objects of type `T` represented by a `Future<T>` is that the object must define a copy constructor, denoted by the type signature `T(const T&)`. The copy constructor allows an instance of type `T` to be constructed from any other instance of type `T`, as is done in the `new` expression of the assignment operator. Primitive types have an implicit copy constructor defined. User-defined types must explicitly specify the copy constructor. The assignment operator, in creating and binding a type `T` object, invokes `T`'s copy constructor with the value being assigned to the future. For example, the following fragment invokes the `operator=` method on the object `date` with the value 2001:

```
Future<int> foo = fact(5);    /* invokes foo.operator=(120) */
```

The assignment operator in turn creates an object of type `int`, binding it to the `result` component of the object bound to `future`:

```
future->result = new int(120);
```

The default copy constructor generated by the C++ compiler for integer types is invoked to create a new integer instance.

An alternate overloading of the assignment operator is provided to permit destructive assignment of future objects, denoted by the type signature: `operator=(Future&)`. The assignment of one future object to another results in both future objects sharing the same capsule object, with the appropriate adjustments being made for reference counting.

Type Conversion: Next to assignment, type conversion is the most important operation defined on polymorphic future types. The type conversion operator method `operator T` allows a consumer thread to obtain a value of type `T` by type converting an object of type `Future<T>` to an object of type `T`. The invocation of the type conversion operator results in the creation of a `Consumer` object that will block the consumer thread if the result represented by the future is not yet available.

Implicit type conversion is accomplished by using a future instance of type `Future<T>` wherever a type `T` value is expected. While implicit type conversion greatly facilitates the use of futures, it can result in unexpected type conversions if used naïvely. The most common unexpected implicit conversion occurs when passing a future instance to a procedure expecting a value of type `T`. Since argument evaluation

is applicative-order, the type conversion operator will implicitly be applied to the future instance in an attempt to obtain a value of type T. For example, passing a `Future<int>` object to the function `foo` defined by the following type signature may cause the procedure call to block pending the conversion of the `Future<int>` to a value of type `int`.

```
void foo(int x);
Future<int> future;
:
foo(future);    /* invocation may block */
```

If the future is not yet ready, the call to `foo` is equivalent to the following sequence:

```
int x = future.operator int();    /* explicit type conversion */
foo(x);
```

The right side of the assignment forces the conversion of the future to an integer, possibly blocking before assigning the result to the temporary. The call to `foo` is then made with the integer value that is bound to `x`.

The static typing of C++ forces the explicit use of `Future<T>` types in procedure type signatures to effect call-by-need semantics. If lazy evaluation of a future is required, then a specific `Future<T>` type should be declared as a formal parameter. For example, the signature for the `foo` procedure can be rewritten as follows:

```
void foo(Future<int> x);
```

The body of `foo` can then effect the type conversion of the future denoted by `x` at the appropriate time.

Dereference: The dereference operator method `operator*` has the same semantics as the type conversion operator. Applying the dereference operator to a future instance results in an object of type T. For example, applying the dereference operator to a `Future<char*>` instance representing a future for a string will cause the formatted print statement in the following code fragment to block until the string represented by the future has been read:

```
Future<char*> futureString;
printf("value read is %s", *futureString);
```

Member Access: The member access operator method `operator->` allows a consumer thread to manipulate the result value of a future *through* the future by obtaining a constant pointer of type T to the result. The `const` attribute applied to the result value of type `T*` indicates that the pointer returned by the member access operator is restricted to invoking only `const` operations. For example, assuming a type `String`, on which is defined a `length` operation with the `const` attribute, the following code fragment allows the `length` operation to be called through the future without first explicitly obtaining the `string` object:

```
Future<String> futureString;
:
int len = futureString.length();
```

The restriction permitting access to only constant operations through the future is necessary to maintain the integrity of the future. Without the restriction, it would be possible to destroy the object encapsulated by the future using an explicit call to the `delete` operator defined by default on all C++ objects.

2.2.3 Predicates and Relations

Like the operator methods, the predicate and relation methods are defined inline. Unlike the operator methods, predicate and relation methods are “constant” operations. Constant operations do not require synchronization control because they do not require that the future dereference the encapsulated value. Predicates and relations may be invoked at any time and do not cause the calling thread to block.

Existence: The existence predicate `exists` is a constant member function that is used to determine whether or not the future can be evaluated (by type conversion, dereferencing, or member access) without blocking.

Comparison: The equality predicate `operator==` is used to compare two instances of a `Future<T>` type. The inequality predicate `operator!=` is the dual of the equality operator method. The comparison is between two `Future<T>` objects and not the future values they denote. For example, the conditional in the following code fragment will immediately evaluate to `FALSE`, regardless of whether or not the connect operation has returned a result.

```
Future<int> future1 = connect(...);    /* async network connect */
Future<int> future2 = connect(...);
:
if (future1 == future2)               /* evaluates immediately to FALSE */
```

Trying to compare two incompatible future types, e.g., `Future<foo>` and `Future<bar>`, is an type error that will be caught at compile time.

To compare a future instance with a value of the same type as that encapsulated by the future, evaluation of the future must first be done by applying either the type conversion operator or the dereference operator. Continuing the above example, the following conditional evaluates to `TRUE` if both future instances refer to the same integer value:

```
if ((int) future1 == *future2)
...
```

3 First-Class Function Objects

An important feature of the `Future<T>` type is that it can be used as the parameterized result type of a first-class function object, called here a *lambda object* because of the obvious correspondence to a lambda abstraction in the lambda calculus. The principal idea is that the `Future<T>` type template in conjunction with a `Lambda<T>` type template, representing a lambda abstraction, are sufficient mechanisms for expressing concurrent computations. A `Lambda<T>` type implements a general closure for a computation, all that is necessary to effect concurrent computation is a mechanism for a delayed result value and the ability to fork a thread. The closure represents the lexical bindings required for a computation to proceed as an independent thread. In using threads in Modula-3, a `closure` type is defined that is used as the argument to an `apply` method that results in the execution of a separate thread of control. As can be seen in Figure 5, a `Lambda<T>` object is a more generalized representation of a closure type, that allows ad hoc specialization by the programmer. The programmer defines an overloading of the “function call” operator, `operator()`, to effect application, or substitution of arguments for bound variables defined as instance variables in the closure.

For example, Figure 5 depicts a type `Lambda<Future<T>>` that defines a `Lambda` object whose `result` has type `Future<T>` and whose type conversion operator return a `Future<T>`. This type is generated by the compiler when the `Lambda<T>` type is parameterized with a `Future<T>` type. The `resultis` macro

```

template<class T> class Lambda {
protected:

    Future<T>    result;           /* "future" type T result value */
                                5
    virtual void lambda() = 0;    /* no default implementation */

    static void eval(Lambda<T>* obj) { obj->lambda(); }

public:                                10

    inline Lambda<T>& fork() {
        thread_create(eval, this);
        return *this;
    }                                    15

    inline operator Future<T> () { return result; }
};

/* result value return macro */
                                20
#define resultis(x)    do { result = x; return; } while (FALSE)

```

Figure 5: Parameterizing a Lambda using a Future.

used by the `lambda` method for assigning a result value invokes the future assignment operator to bind the result to the future.

As depicted in Figure 6, a concurrent factorial can be defined by providing a `Future<int>` type as the type parameter to the inherited `Lambda<T>` class. The inline `fact` function creates a thread that will schedule the `eval` method for execution and return a `Future<int>` as a result. The concurrent factorial is simply invoked as:

```

Future<int> future = fact(n);
:
int result = future;

```

4 Other Future Mechanisms

The polymorphic future type presented in the previous section is compared with other future mechanism; notably: the future primitive in MultiLisp, the promise type in Argus, the Cbox in Concurrent Smalltalk, the future functor in Concurrent ML, the return-to-future mechanism in Mentat, and the *future ref* in ES-Kit.

4.1 Futures in MultiLisp

Halstead introduced the concept of a future as a control abstraction in MultiLisp, a concurrent Scheme implementation [9]. Futures are similar in concept to the `delay` and `force` constructs that together support lazy evaluation in sequential Scheme [4]. A `delay` expression evaluates to a lambda expression that can then be used as an argument to the `force` procedure to cause the evaluation of the delayed expression; for example:

```

(define foo (delay (factorial 50)))

```

```

class Fact : public Lambda<int> {
private:

    int n;

    void lambda() {
        if (n > 1) {
            Fact f;
            result = n * f(n-1);
        }
        else
            result = 1;
    }

public:

    /* "apply" method */
    inline Lambda<int>& operator()(int i) { n = i; return fork(); }
};

/* concurrent factorial "applicator" */
inline Future<int> cfact(int n) { return (new Fact)->operator()(n); }

```

Figure 6: A Concurrent Factorial using a Future.

```

:
(... (force foo) ...) ;; evaluate (lambda() (factorial 50))

```

The expression `(factorial 50)` is not evaluated, but instead made into an implicit `lambda` that is maintained in the environment and eventually evaluated by `force`.

A `future` expression in MultiLisp is used in a manner similar to `delay`:

```
(define foo (future (factorial 50)))
```

The result of a `future` expression is a value representing the parallel evaluation of the expression `(factorial 50)`, rather than a `lambda` expression that when forced, will evaluate to `50!`. Thus, futures in MultiLisp provide the opportunity for expressing a high degree of parallelism in computing MultiLisp expressions. For example, construction of a *conscell* data structure using the `cons` function may be expressed as:

```
(define conscell (cons (future E1) (future E2)))
```

Since each `future` expression generates a concurrent computation, returning an immediate future value as the result, the `cons` expression immediately constructs a two element `conscell` containing two future values. When the respective evaluations of expressions E_1 and E_2 are complete, the futures in the `conscell` denoted by `foo` are implicitly replaced with the values of the concurrently computed expressions. Should a future value be referenced in another expression before the computation represented by the future has completed; e.g., `(car foo)`, the evaluation of that expression blocks until the future is replaced with its value by the corresponding parallel computation.

In MultiLisp, once a computation is started for a future expression it is not possible to force the computation to abort. The computation will proceed to completion, even though the eventual result might not be needed. A more powerful control abstraction would permit the treatment of a future as an abstract data type allowing the specification of operations (e.g., `cancel`) on futures themselves and not the values they denote.

4.2 Promises in Argus

Futures in MultiLisp motivated Liskov to introduce the concept of a *promise* in conjunction with *call-streams* in Argus [18]. Argus is a language and a run-time system for programming distributed applications [14, 15]. The language is a direct descendant of CLU [16], both in its syntax and semantics. Unlike the dynamically typed MultiLisp, Argus is statically typed; hence, promises are strongly typed. A call-stream is an abstraction that combines the semantics of remote procedure call and message sending by allowing non-blocking procedure invocations. As with a future in MultiLisp, a promise is a structure representing the future result value of a concurrent computation. Unlike a future, a promise is able to maintain information about exceptions that might have been raised by the concurrent computation.

Syntactically, a promise is declared in a type expression that is consistent with a corresponding procedure type signature:

```
factorial:proc (n:int) returns(int) signals(overflow)
P = promise returns(int) signals(overflow)
```

A key feature of a promise is that it is consistent with the abstract data type concepts that CLU and its derivatives are founded upon—a promise is an instance of an abstract data type that is parameterized with type information related to the type of the result value encapsulated by the promise. The specification of a promise indicates the type of the result value of the corresponding procedure and any exception types. To claim a result from a promise, the `claim` operation is invoked on an instance of a promise type:

```
p:P := fork factorial(50)
:
x:int := P$claim(p)
```

If the result of the concurrent computation represented by `factorial` has not been produced at the time of the call to `claim`, the calling process blocks. If an exception is raised, the exception handling code for type `overflow` is automatically invoked.

4.3 Cboxes in Concurrent Smalltalk

Concurrent Smalltalk, a concurrent extension of sequential Smalltalk, employs a *Cbox* class to facilitate asynchronous method invocation [26]. Method invocation in Concurrent Smalltalk is expressed using a messaging paradigm where messages denoting methods defined in a class definition are sent to instances of the class to effect computation. An asynchronous invocation is distinguished from a synchronous invocation by the special “&” symbol appended to the name of a message sent to an object. For example, the following code fragment asynchronously sends the `factorial` message to the number object represented by the symbol `50`. The expression returns a Cbox object that is bound to the locally declared, dynamically typed `future` identifier.

```
|future|
:
future ← 50 factorial&.
```

Similar to previous future mechanisms, a Cbox object is the recipient of the eventual result of a concurrent computation. A result value is obtained from a Cbox by sending the `receive` message to a Cbox object. The `receive` method blocks if a result value has not yet been computed and bound to the Cbox object:

```
result ← future receive.
```

An elegant aspect of the first-class nature of Cboxes is that they can be used to implement and/or-synchronization. Cboxes may be placed into collections and a thread may then wait for all or some of the Cboxes in the collection to have result values pending using various and/or-synchronization messages (e.g., `receiveAnd:` and `receiveOr:`) sent to a Cbox instance representing the collection.

4.4 Future Functors in CML

A polymorphic future may be defined in Concurrent ML using either synchronous channels directly, or write-once condition variables [20]. The utility of a future in CML is questionable since communication is primarily synchronous. The advantage of a future mechanism arises when there is asynchronous communication. Hence the future functor in CML is more a demonstration of what can be achieved using the features of CML than a useful programming construct.

4.5 Return-to-Future in Mentat

Mentat uses asynchronous invocations and a vaule-based return form, which is similar to a future [8]. The key characteristic of the Mentat return-to-future (`rtf`) is that the future object in mentat is implicit. The programmer simply declares the invocation and names the variable to receive the result value. Since the invocation is asynchronous, the value of the return variable is initially undefined. The Mentat compiler has the responsibility of generating the code that allows continued execution of the sender until such time as the sender actually needs to evaluate a return value. Like futures in MultiLisp, the variable to receive an eventual result may be passed as an argument to a procedure without causing evaluation of the future, and possible blocking. Mentat carefully records all objects waiting on a future so that when a result does appear, the `rtf` mechanism delivers the result value to all objects awaiting the value.

4.6 Future References in ES-Kit

An interesting application of the concept of futures in an imperative setting is the *future ref* used in the Experimental Systems Kit (ES-Kit), a distributed kernel and run-time system for distributed and parallel programming in C++ [13].⁵

A future ref in an ES-Kit application is an instance of a system defined `FutureRef` class; hence, a future ref object is a first-class type. A `FutureRef` instance is returned as the result of an asynchronous method invocation using an overloading of a non-standard *method call* operator, denoted by the symbol `->()`. In C++, operator overloading is a form of *ad hoc* polymorphism that allows a different implementation to be associated with certain operations when applied to instances of a class. The method call operator, when applied to an instance of a special system class, effects an asynchronous procedure call that immediately returns a `FutureRef` instance as a result. The type signature for this operation is as follows:

```
FutureRef operator->()(int id, int len, ...);
```

The arguments are a method identifier, the length of the argument vector, and an arbitrary argument vector.⁶ Interestingly, the semantics given to this operator in ES-Kit are not consistent with the semantics defined by the C++ language [7]. The method call operator has no equivalent in standard C++. The ES-Kit method call operator is a combination of the standard C++ unary *member access* operator `->` and the binary *function call* operator `()`. The access operator is most often used to implement “smart pointers” [22]; whereas, the function call operator is often used to implement an iterator as part

⁵The GNU C++ compiler [25] was begun by Tiemann at MCC as part of the Experimental Systems Project in an effort to make C++ usable for distributed and parallel programming.

⁶The method identifier corresponds to an entry in the method dispatch table for the object to which the operator is applied (i.e., the `vtable`).

of an *iteration abstraction* [17]. To obtain the binary method call operator, the ES-Kit environment requires a modified compiler; hence, the language is technically a semantic extension of C++.

If an object is defined as an instance of a user-defined class that is derived from the special system class defining the overloaded operator, then all method invocations in the derived class will return future refs as result values. In this regard, a future ref is similar to a MultiLisp future or an Argus promise in that the future type represents a handle for a concurrent computation. In C++, the invocation of a method of some object pointed to by `obj` and returning a result value of type `T` is specified as:

```
T result = obj->method(a1, a2, ..., an);
```

Under normal C++ procedure call semantics, a stack context is established for the method, with the usual pushing of argument values (including the value of `obj` as an environment binding context and return address.⁷) The call to `factorial` is then made and, in this case, an integer value is returned (probably through a register) and assigned to the storage location denoted by `result`.

A future ref, in conjunction with the overloading of `operator->()`, provides an asynchronous procedure call semantics. An invocation is specified as:

```
FutureRef future = obj->factorial(50);
:
int result = future;    /* implicit type conversion */
```

The call to `factorial` initiates an asynchronous invocation and immediately returns a `FutureRef` instance that is bound to `future`. At some later point in the computation, the future ref is implicitly type converted to an `int` by assignment to `result`.

A limitation with future refs is that they are not general polymorphic types. Type conversion is limited to the set of primitive language types `char`, `int`, `float`, etc., and pointers to these types. It is straightforward to define the small set of type conversion operators required to handle the primitive language types. Arbitrary user-defined types however are not handled as elegantly as one would like—the untyped `void*` pointer is used to allow arbitrary user-defined types, with the burden placed on the programmer to explicitly perform the appropriate type conversion at run-time.

Unlike futures in MultiLisp or promises in Argus, a future ref in ES-Kit is first-class. Using an overloaded equality relation defined as an operation on the type, it is possible to determine whether or not two futures refs, and not the values they denote, are equivalent. The comparison of two futures does not result in a blocked computation should the value of either future not yet be computed. For example, the following conditional tests whether or not the two future ref instances refer to the same concurrent computation, and not whether the values they denote are equivalent.

```
if (future1 == future2)
...

```

If it is really the values that are to be compared, then explicit type conversion can be used to force the futures to be evaluated. In the following example, the evaluation of two future refs to the integers they represent is forced by an explicit type conversion. The equality relation is then applied to two integer values when the computations denoted by the futures are both completed.

```
if ((int) future1 == (int) future2)
...

```

In contrast, the MultiLisp expression `(eq? future1 future2)` will block if the computations denoted by `future1` and `future2` have not both produced a value by the time the equality expression is evaluated. Since MultiLisp is dynamically typed, there is no equivalent to explicit type conversion; instead, a special equality relation and symbol must be defined to compare whether or not two futures refer to the same computation.

⁷The environment binding context is the first argument on the stack and is represented by the keyword `this`.

Table 2: Comparison of Future Mechanism Type Characteristics.

Future Mechanisms	Type Characteristics			
	Resolution	Conversion	First-Class	Polymorphic
MultiLisp Futures	dynamic	implicit	no	yes
Argus Promises	static	explicit	no	no
C-Smalltalk Cboxes	dynamic	implicit	yes	yes
ES-Kit Future Refs	static	explicit/implicit	yes	no

4.7 Summary

Table 2 summarizes the type characteristics of each of the future mechanisms just described. The characteristics of interest are: type resolution, type conversion, first-class properties, polymorphism. Type resolution is distinguished by whether or not the type of a value is determined at run-time (dynamic) or compile-time (static). Type conversion is categorized as either implicit, explicit, or both. A future is considered a first-class type if it is possible to specify user-defined operations as part of its representation. A future is polymorphic if it is possible to define arbitrary future types. In general, dynamic type resolution implies polymorphism.

5 Future Work

The use of template classes in C++ allows the creation of polymorphic future types and polymorphic capsule types. An interesting area for future work is to implement the future, capsule, monitor, and mutex abstractions in Ada and Modula-3. Both Ada and Modula-3 provide generic types, and unlike C++, provide built-in support for multitasking. Ada implements a *CSP*-style rendezvous tasking model; Modula-3 implements a lightweight threads model like the one assumed here. Modula-3 also provides built-in support for garbage collection, which would facilitate the storage management of future types.

A useful experiment would be to compare and contrast all three implementations in terms of their structure and performance. The original Ada definition does not define an inheritance mechanism, so a non-inheritance based compositional relationship between capsules and monitors would have to be designed.⁸ Modula-3 provides a subtyping mechanism, so the structure here should be directly translatable.

A more theoretical avenue of research is to explore the semantics of polymorphic future types from a denotational perspective. Continuations have mostly been explored in a sequential language setting, primarily in the context of Scheme. Continuations have been used to implement coroutine semantics [10], which is a more limited form of concurrency than assumed by future types. Given the close relationship between continuations and futures in asynchronous invocation semantics, there is the potential for achieving some useful theoretical results relating type conversion and synchronization in the context of polymorphic future types.

References

- [1] Brian N. Bershad. The PRESTO user's manual. Technical Report 88-01-04, University of Washington, Department of Computer Science, 1988.
- [2] Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. PRESTO: A system for object-oriented parallel programming. *Software—Practice and Experience*, 18(8):713–732, August 1988.

⁸The forthcoming Ada-9X definition does define an inheritance mechanism.

- [3] Brian N. Bershad, David D. Redell, and John R. Ellis. Fast mutual exclusion for uniprocessors. In *Fifth ACM Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 223–233, October 1992.
- [4] William Clinger and Jonathan Rees. *Revised Report on the Algorithmic Language Scheme*, 1992. Version 4.
- [5] Eric C. Cooper and Richard P. Draves. C threads. Technical Report CMU-CS-88-54, Carnegie Mellon University, School of Computer Science, September 1990.
- [6] Jack W. Davidson and Anne M. Holler. Subprogram inlining: A study of its effects on program execution time. *IEEE Transactions on Software Engineering*, SE-18(2):89–102, February 1992.
- [7] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.
- [8] Andrew S. Grimshaw. Easy-to-use object-oriented parallel processing with Mentat. *IEEE Computer*, pages 39–51, May 1993.
- [9] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.
- [10] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Obtaining coroutines with continuations. *Computer Languages*, 11(3/4):143–153, 1986.
- [11] Richard L. Hudson, J. Eliot B. Moss, Amer Diwan, and Christopher F. Weight. A language independent garbage collector toolkit. To appear, 1991.
- [12] IEEE Technical Committee on Operating Systems. *Threads Extension for Portable Operating Systems*, February 1992. POSIX P1003.4a/D6.
- [13] Bill Leddy and Kim Smith. The design of the experimental systems kernel. In *Proceedings of the Conference on Hypercube and Concurrent Computer Applications*, Monterey, CA, 1989.
- [14] Barbara Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.
- [15] Barbara Liskov, Dorothy Curtis, Paul Johnson, and Robert Scheifler. Implementation of Argus. In *Eleventh ACM Symposium on Operating System Principles*, pages 111–122, November 1987.
- [16] Barbara Liskov and et al. *CLU Reference Manual*. Springer-Verlag, 1981.
- [17] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Press, 1986.
- [18] Barbara Liskov and Luiba Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the SIGPLAN’88 Conference on Programming Language Design and Implementation*, pages 260–267, June 1988.
- [19] Frank Mueller. A library implementation of POSIX threads under UNIX. In *Proceedings of the 1993 USENIX Winter Conference*, pages 29–41, January 1993. To appear.
- [20] John H. Reppy. *Concurrent Programming with Events: The Concurrent ML Manual*. AT&T Bell Laboratories, February 1993. version 0.9.8.
- [21] Richard M. Stallman. *Using and Porting GNU CC*. Free Software Foundation, 1992. Version 2.2.
- [22] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, second edition, 1986.

- [23] Sun Microsystems. *Sun Release 4.0 Programmer's Reference Manual*, March 1990. Section 3L: Lightweight Processes Library.
- [24] Sun Microsystems. *Sun System Overview*, March 1990. Lightweight Processes.
- [25] Michael D. Tiemann. *User's Guide to GNU C++*. Free Software Foundation, 1992.
- [26] Yasuhiko Yokote. *The Design and Implementation of Concurrent Smalltalk*. World Scientific, 1990.