Loci Programming Language

Stephen Cross

Overview

- Systems level language
- Influenced by C and particularly C++, but incorporates aspects of many languages
- Very close compatibility with C
- Carefully designed to match run-time overheads of C++

Hello World (C style)

```
int main(int argc, char** argv) {
    printf(C”Hello world!\n”);
    return 0;
}
```

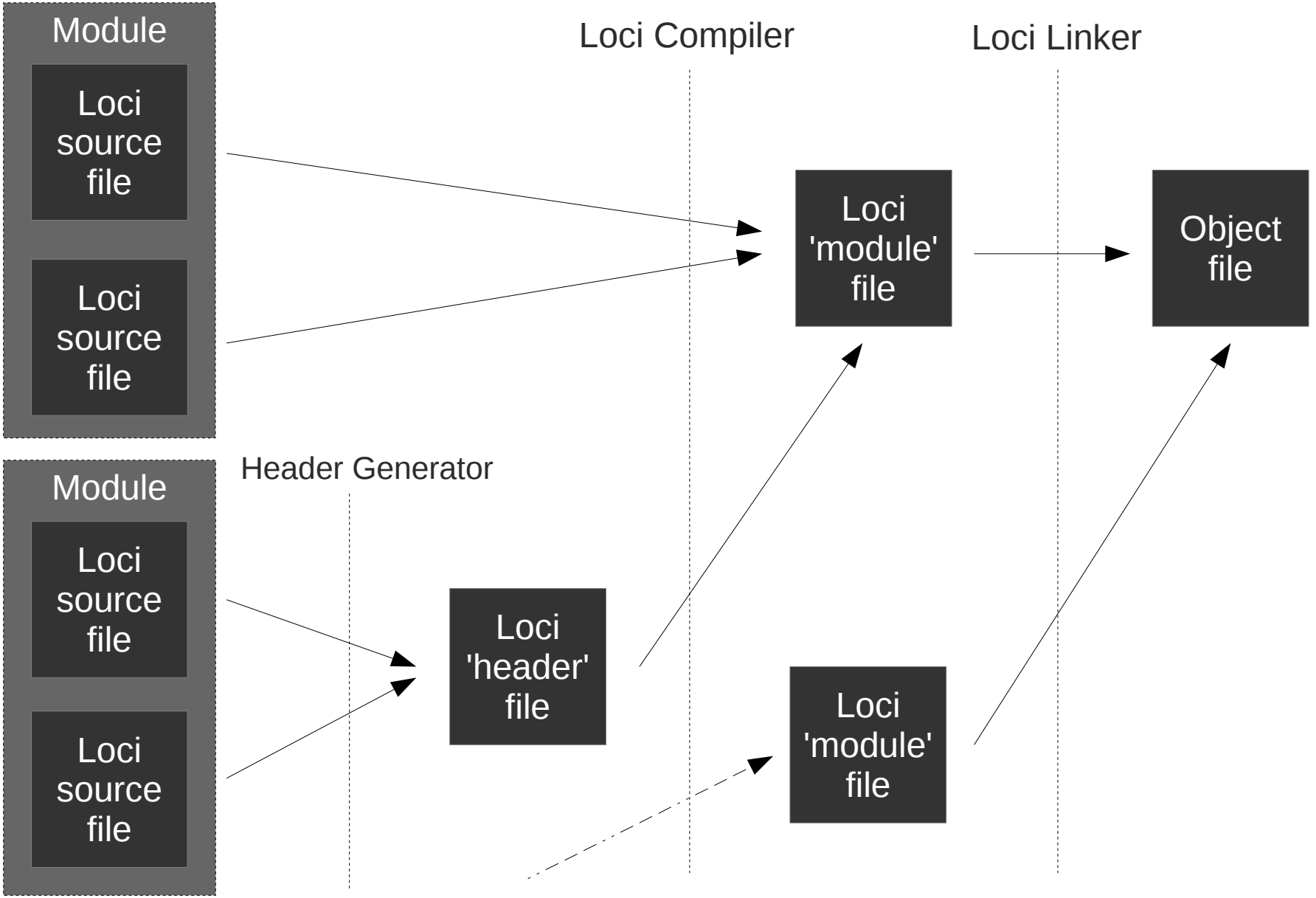
Hello World (Loci style)

```
int main(int argc, char** argv) {
    std::print(”Hello world!\n”);
    return 0;
}
```

Why?

- C++ has problems:
  - Overly complex syntax and semantics
  - Easy to trigger undefined/unspecified behaviour
  - Valuable features missing
  - Excessive verbosity
- End the need to use C for APIs:
  - Implementation specification means binary compatibility between compilers
  - Enhanced encapsulation over C++
  - Callable from all C-compatible languages

Compilation Strategy



Classes

- Can generate declaration from definition
- *const* only needed in declaration
- Private variables never mentioned in declaration
- Destructor never mentioned in declaration
- Avoid writing class name more than once

Declaration

```
class ExampleClass {
    static default();

    int getValue() const;

    void setValue(int value);
}
```

Definition

```
class ExampleClass (int value) {
    static default () {
        // Constructor.
        return @(0);
    }

    ~ {
        // Destructor.
    }

    int getValue() {
        return @value;
    }

    void setValue(int value) {
        @value = value;
    }
}
```

Move Operations

- **C++: *`copy`* by default, *`move`* if supported**
- **Loci: *`move`* by default, *`copy`* if supported**
- Adds *`move`* operator:

```
ExampleClass value0 = …;
ExampleClass value1 = move value0;
```

- *`move`* returns r-value; makes l-value *`empty`*
- *`empty`* objects are just zeroes
- No destructor is run for an *`empty`* object
- *`empty`* objects can be filled by assignment

Move Operations (1)

C++

```
template <typename T>
class auto_ptr {
      public:
            auto_ptr()
                  : ptr_(NULL) { }

            auto_ptr(T* ptr)
                  : ptr_(ptr) {
                  assert(ptr != NULL);
            }

            auto_ptr(auto_ptr<T>& p){
                  ptr_ = p.ptr_;
                  p.ptr_ = NULL;
            }

            auto_ptr<T>& operator=(const auto_ptr<T>& p) {
                  If (&p != this) {
                        delete ptr_;
                        ptr_ = p.ptr_;
                        p.ptr_ = NULL;
                  }
                  return *this;
            }
            ~auto_ptr() {
                  delete ptr_;
            }

      private:
            T* ptr_;

};
```

Loci

```
template <typename T>
class auto_ptr (T* ptr) {
    static null() {
        return @(null);
    }

    static default(T* ptr) {
        assert(ptr != null);
        return @(ptr);
    }

    ~ {
        delete @ptr;
    }
}
```

Move Operations (2)

- In C++, objects are tied to their location
- In Loci, objects are ***transient***; they move between l-values
- Objects can have the ***role*** of l-value
- *`lval`* keyword enables *operator overloading* for lvalue operations
- ***Single responsibility principle***

lval

```
class int_lvalue {
    int* opAddress();
    void opAssign(int newValue);
    int opMove();
    int& opReference();
}

void example(lval int_lvalue lvalue) {
    // Calls lvalue.opAssign(1);
    lvalue = 1;
    // Calls lvalue.opMove();
    int movedVal = move lvalue;
    // etc.
}
```

Primitives are Objects

Using primitives as objects

```
float minusQuarter = -0.25f;
float plusQuarter = minusQuarter.abs();
float zero = plusHalf.floor();
float plusHalf = plusQuarter.sqrt();
```

- Primitives types are object types in Loci
- Generally this is just syntactic sugar...
- ...however it can be used with polymorphism (next)

Compiler Internal Declaration

```
__primitive float {
    float abs() const;
    float floor() const;
    float sqrt() const;
    // etc.
}
```

Polymorphism - Structural Typing

Class

```
class ExampleClass {
    static default();

    int getValue() const;

    void setValue(int value);
}
```

Interface

```
interface ExampleInterface {
    int getValue() const;

    void setValue(int value);
}
```

- No need for explicit *A implements B*
- If a class has the required methods, then it implements the interface
- Useful if defining interfaces **after** classes
- Works for any type (e.g. *int* or *float*)

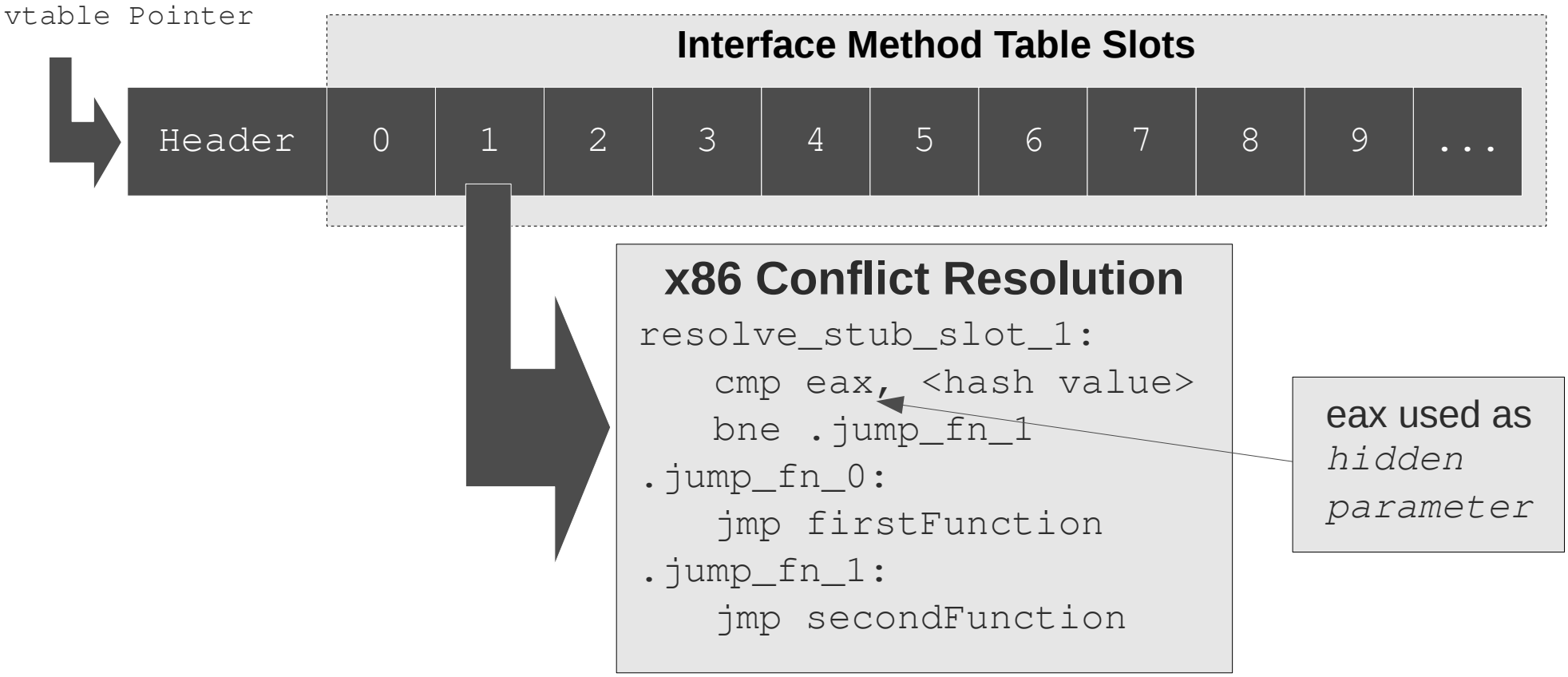
Polymorphism

```
int postIncrement(ExampleInterface& object) {
    int value = object.getValue();
    object.setValue(value + 1);
    return value;
}

void example() {
    auto instance = ExampleClass();
    int value = postIncrement(instance);
    std::print("Was %0, now %1.\n".format(
        value, instance.getValue()));
}
```

Polymorphism Implementation

- Fixed size hash table with stubs to resolve conflicts (using a *`hidden parameter`*)
- Simply load the function pointer at slot *`HASH(name) % NUM_SLOTS`* and call it
- Relative to C++ virtual call:
  - In case of collision: Very low overhead (see x86 code below for an example)
  - No collision: No overhead (other than assigning *`hidden parameter`*)



Templates

- Generalise a class or function
- Not for compile-time evaluation; use *`forceeval`* for that
- Interfaces specify type parameter *`constraints`*

```
template <typename T>
interface Copyable {
   T copy() const;
}

template <typename T: Copyable<T>>
T copyValue(const T& value) {
   return value.copy();
}
```

Compile-Time Evaluation

- Similar to run-time evaluation
- *`forceeval`* forces compile-time evaluation

```
// Admittedly bad way to
// calculate fibonacci numbers.
int fib(size_t i) {
   return i == 0 ? 0 :
             i == 1 ? 1 :
                fib(i - 2) + fib(i - 1);
}

void example() {
   int fib30 = forceeval fib(30);
   std::print("fib(30) = %0".format(fib30));
}
```

Algebraic Datatypes

- Influenced by functional languages
- Effectively a combination of enum, struct and union

```
template <typename T>
datatype Maybe = Something(T value) | Nothing;

template <typename T>
void whatIsIt(Maybe<T> maybe) {
   switch (maybe) {
      case Something(_) {
         std::print(“It's something!”);
      }
      case Nothing {
         std::print(“It's nothing...”);
      }
   }
}
```

Algebraic Datatypes (1)

- Can *split* functions by *pattern matching*

```
using namespace std;

interface stringable {
    string toString();
}

string toString(Nothing) {
    return “Nothing”;
}

template <typename T: stringable>
string toString(Something<T> something) {
    return “Something(%0)”.format(something.value);
}

void example() {
    print(“Got %0”.format(Something<int>(42)));
}
```

BONUS

Virtual Template Parameters

- Template `typename` only accepts *`concrete`* type parameters
- *`virtual typename`* accepts concrete **or** abstract parameters

```
interface BasicType {
   void doSomeThings();
}

interface AdvancedType {
   void doSomeThings();
   void doAllThings();
}

template <virtual typename T: BasicType>
void callDoSomeThings(T& object) {
   object.doSomeThings();
}

void example(AdvancedType& object) {
   callDoSomeThings(object);
}
```

BONUS

Virtual Template Parameters (1)

- Enables covariant and contravariant implicit casts
- Useful for reference types (Loci has *smart references*)

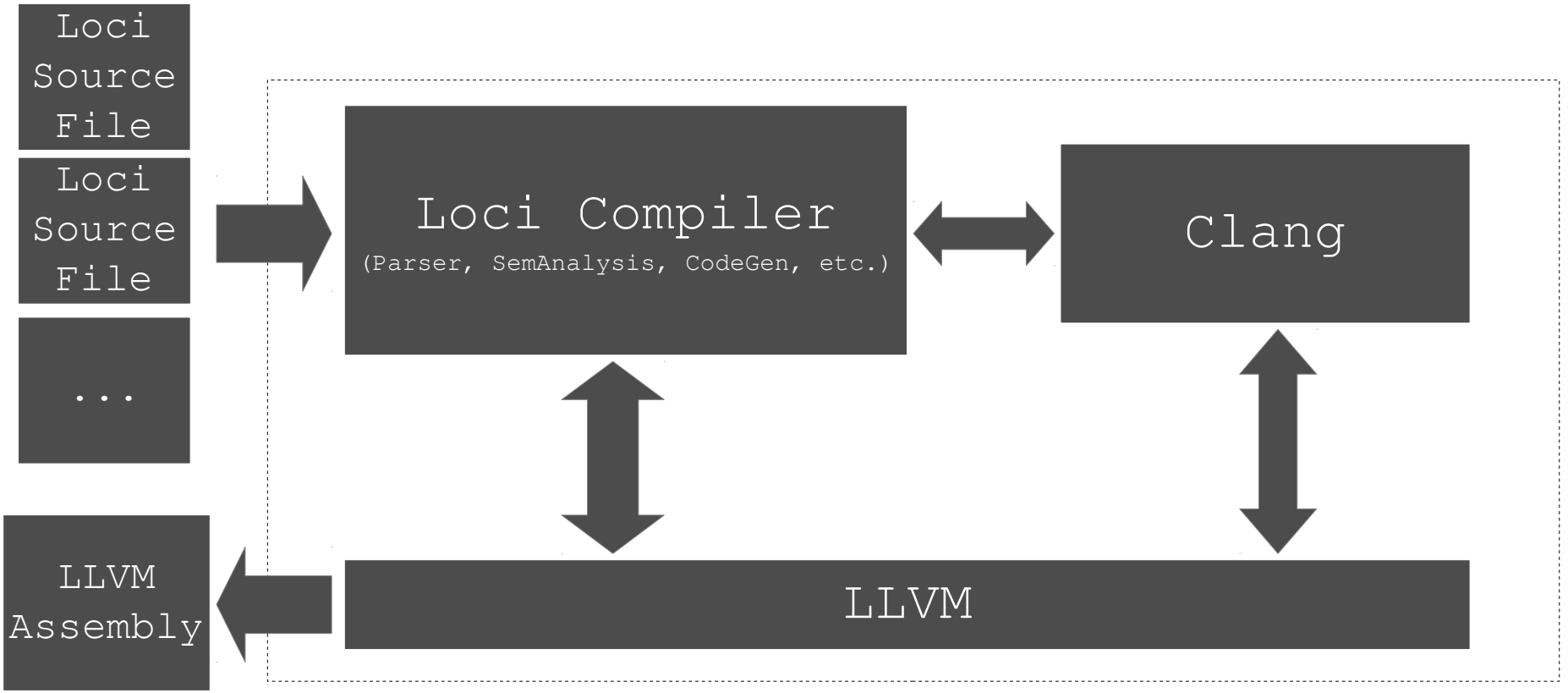
```
template <virtual typename T>
class cov {
    T& get() const;
}
template <virtual typename T>
class ctv {
    void set(T&);
}

// Covariance.
cov<BasicType> f0(cov<AdvancedType> arg) {
    return arg;
}

// Contravariance.
ctv<AdvancedType> f1(ctv<BasicType> arg) {
    return arg;
}
```

Status

- Around half of features implemented
- Compiler currently generates LLVM assembly
- Clang used for target info (e.g. `sizeof(int)`)



Questions?