Kai: Software Overview

Samuel Grant Dawson Williams

October 1, 2010

Abstract

Kai is an experimental interpreter that provides explicit control over the compilation process. It can generate optimised code at run-time in order to exploit the nature of the underlying hardware. It is a unique exploration into world of dynamic code compilation, and the interaction between high level and low level semantics.

Contents

1 Installing Kai

Kai uses **CMake** for compilation. It can be compiled on Linux and Mac OS X. It requires **libgc** and **LLVM**. The source code of Kai includes a copy of LLVM 2.7 as an external dependency.

```
$ git clone http://github.com/ioquatix/kai.git
Cloning into kai...
remote: Counting objects: 374, done.
remote: Compressing objects: 100% (364/364), done.
remote: Total 374 (delta 256), reused 0 (delta 0)
Receiving objects: 100% (374/374), 99.75 KiB | 103 KiB/s, done.
Resolving deltas: 100% (256/256), done.
```

1.1 Installing LLVM

```
$ cd kai
$ git submodule update --init
$ cd ext/llvm
$ mkdir build
$ cd build
$ cmake -DCMAKE_INSTALL_PREFIX=/usr/local/llvm-2.7 -DCMAKE_BUILD_TYPE=Release ..
$ sudo make install
```

This will install **LLVM** into '/usr/local/llvm-2.7'.

1.2 Installing Kai

```
$ cd kai
$ mkdir build
$ cd build
$ cmake ..
$ sudo make install
```

This will install **kai** into '/usr/local/bin'.

2 Using Kai

After starting kai, you will see on the command line a prompt:

```
kai>
```

From this point, you can type expressions, and they will be evaluated. The return value will be printed, and the previous expression will be saved in a special variable '_'.

```
kai> [10 + 20]
30
kai> [_ * 100]
3000
```

Kai currently supports a limited set arithmetic operations including + (sum), − (subtract), ∗ (product) and % (modulus).

```
kai> [[10 * 100] % 11]
10
kai> [10 * 2 5 10]
1000
kai> [10 + 2 5 10]
27
kai> [x - 100]
-90
```

3 Syntax

3.1 Strings

A string represents a sequence of characters. A character is one or more bytes as defined by the encoding. By default, Kai expects strings to be encoded using UTF-8, and does not have provisions for other encodings.

A string is represented literally by data enclosed between two quotation marks. Internal quotation marks can be escaped using a backslash, along with several other characters such as tab, newline and null characters.□Bytes can be inserted using hexadecimal escape sequences.

Listing 1: Strings

```
kai> "Hello\nWorld!"
"Hello
World!"
kai> "Hello\t\tWorld!"
"Hello        World!"
kai> "Apples \x26 Oranges"
"Apples & Oranges"
```

3.2 Numbers

Basic numbers without a decimal point are considered integers. These can be both positive and negative.

Listing 2: Integers

```
kai> -55
-55
kai> 897
897
```

Numbers with a decimal point are current unsupported.

3.3 Symbols

Symbols are sequences of characters which represent an identifier. They are different from strings in the sense that they require no quotation marks, and internally a symbol also has an associated hash value.

Listing 3: Symbols

```
kai> `ThisIsASymbol
ThisIsASymbol
kai> [`ThisIsASymbol hash]
1291
kai> [`hash hash]
420
```

3.4 Cells

A Cell is a basic structure of recursion. A Cell consists of a head and a tail, and is, in traditional literature a single link in a singly linked list. The head of a cell is the value it contains, and the tail of the cell is the next cell in the list, or null.

A cell can represent a multitude of different data structures including lists, trees and graphs.

Listing 4: Cells

```
kai> `(10 apples and 20 oranges)
(10 apples and 20 oranges)
kai> [`(10 apples and 20 oranges) head]
10
kai> [`(10 apples and 20 oranges) tail]
(apples and 20 oranges)
```

3.5 Values

A value expression is a short-hand notation for giving the value of an operand rather than the result of its evaluation. It is done using the back-tick character before an expression.

Listing 5: Values

```
kai> ``bob
(value bob)
kai> `bob
bob
kai> bob
nil
```

3.6 Calls

A call expression is a short-hand notation for method dispatch on a given object. Because the semantics of this operation are non-trivial, having a syntactic expression for this type of function is preferable.

Listing 6: Calls

```
kai> [`orion hash]
551
kai> `[`orion hash]
(call (value orion) (value (hash)))
```

3.7 Blocks

Blocks are collections of code. They are are syntactic sugar to enhance reading code.

Listing 7: Blocks

```
kai> {[(this) set `x 10] [x return]}
10
kai> `{[(this) set `x 10] [x return]}
(block (call (this) (value (set (value x) 10))) (call x (value (return))))
```

4 Examples

4.1 Fibonacci Numbers

```
[(this) set 'fib (lambda '(x) '{
    (if (or [x == 0] [x == 1])
        (return x)
        (return [(fib [x - 1]) + (fib [x - 2])])
    )
})]
```

If we use this function, we can calculate Fibonacci numbers by recursive method. This method is computationally expensive, so don't choose $x > 20$.

```
kai> (fib 10)
55
```

4.2 Greatest Common Divisor

```
[(this) set 'gcd
    (lambda '(a b) '{
        (if [b == 0]
            (return a)
            (return (gcd b [a % b]))
        )
    })
]
```

If we use this function, we can calculate Greatest Common Divisor of two numbers by recursive method.

```
kai> (gcd 892346 23426)
26
```

4.3 Compiled Greatest Common Divisor

```
[(this) set 'gcd
    (compiler 'gcd (function (int 32) (int 32) (int 32)) '(a b) '{
        (if [b == 0]
            (return a)
            (return (gcd b [a % b]))
        )
    })
]
```

This function is compiled by **LLVM**.

```
kai> (gcd 9018345 241234)
7
```

Kai currently has hardcoded bridge between interpreter and compiler using two integer arguments and integer return type. In the future this will be improved to support any kind of function.

4.4 Double Closure

```
[(this) set 'double
    (lambda '(x)
        '(lambda '()
            '(block
                (update 'x [x * 2])
                (return x)
            )
        )
    )
]
```

A closure over a stack frame allows us to retain state. This state is bound to all functions, and thus allows us to create a basic object model. The implications of this are profound.

```
kai> [(this) set 'd (double 10)]
(lambda{0x100bc3900} '() '(block (update 'x (call x '(* 2))) (return x)))
kai> (d)
20
kai> (d)
40
kai> (d)
80
kai> (d)
160
```

5 Implementation

Kai uses two basic data types for implementing the structure of its language: **Frame** and **Value**. Frames represent the stack and provide the scope for name lookup, while values are the abstract base for all values in the programming language.

Every other part of the language is based on these two concepts.

5.1 Frames

Execution flow is represented by a stack of frames. A frame generally represents a message dispatch of some sort, but this isn't always the case.

Listing 8: class Frame

```
class Frame : public gc {
    protected:
        /// Previous stack frame
        Frame * m_previous;

        /// The scope of the stack frame, if any.
        Value * m_scope;

        /// The original message which created this frame, if any.
        Cell * m_message;

        /// The evaluated function.
        Value * m_function;
        /// The unwrapped arguments.
        Cell * m_arguments;

        /// Given a stack frame, apply the function to the arguments.
        Value * apply ();

        /// For debugging - the depth of the stack.
        unsigned m_depth;
    public:
        /// Create a root level stack frame with a given scope.
        Frame (Value * scope);
        /// Create an intermediate stack frame with a given scope and previous frame.
        Frame (Value * scope, Frame * previous);
        /// Create an intermediate stack frame as above, but with a given message.
        Frame (Value * scope, Cell * message, Frame * previous);

        /// Lookup an identifier using the stack, starting at this frame.
        Value * lookup (Symbol * identifier);
        /// Lookup an identifier as above, but return the frame which defines the value
            .
        Value * lookup (Symbol * identifier, Frame *& frame);

        // Should a message be restricted to a Cell, or is it suitable to be a Value ?
        Value * call (Value * scope, Cell * message);

        /// Return the previous stack frame.
        Frame * previous ();

        /// This function searches up the stack for the current scope.
        Value * scope ();
```

```
        /// Return the message (m o1 o2 o3) if it is defined.
        Cell * message ();

        /// Return the defined function (m), if it is known.
        Value * function ();
        /// Return the operands (o1 o2 o3) if they are given.
        Cell * operands ();
        /// Evaluate the operands in the current stack frame.
        Cell * unwrap ();
        /// Return the arguments if they have been evaluated.
        Cell * arguments ();
};
```

5.2 Values

Values provide several high level semantics for the major paradigms of Kai. This includes:

- The **evaluate** function which executes the value and returns the result.
- The **lookup** function which is used for name lookup.
- The **prototype** function which returns functions associated with the value.
- The **compile** function which is used for the compiler to generate LLVM compiled values.
- The **import** function which imports functionality into an execution context.

Listing 9: class Value

```
class Value : virtual public gc {
    public:
        /// Compare the value with another value. Returns -1, 0, 1 depending on
            comparison result.
        /// If objects cannot be compared, throws InvalidComparison exception.
        virtual int compare (Value * other);

        /// Write the value to the given buffer.
        virtual void toCode(StringStreamT & buffer, MarkedT & marks, std::size_t
            indentation);

        /// Lookup the given identifier. Defers to prototype by default.
        virtual Value * lookup (Symbol * identifier);

        /// A prototype specifies the behaviour of the current value.
        virtual Value * prototype ();

        /// Evaluate the current value in the given context.
        virtual Value * evaluate (Frame * frame);

        /// Compile the value to an llvm Value
        virtual llvm::Value * compile (Frame * frame);

        /// If the value contained is a compiled value, return it unevaluated.
        virtual llvm::Value * compiledValue (Frame * frame);

        /// kai> (sleep <Integer>)
        static Value * sleep (Frame * frame);

        /// The global value prototype.
```

```
        static Value * globalPrototype ();

        /// Import the global prototype and associated functions into an execution
            context.
        static void import (Table * context);
};
```

5.3 Execution Context

The execution context is the core of the Kai interpreter.

Listing 10: Execution Context

```
Table * buildContext () {
    Table * global = new Table;
    global->setPrototype(Table::globalPrototype());

    Integer::import(global);
    Frame::import(global);
    Value::import(global);
    Symbol::import(global);
    Cell::import(global);
    String::import(global);
    Table::import(global);
    Lambda::import(global);
    Logic::import(global);
    Expressions::import(global);

    Compiler::import(global);
    CompiledType::import(global);

    Table * context = new Table;
    context->setPrototype(global);

    return context;
}
```

To execute some source code, firstly we parse it into a value, then we create a root stack frame and use this for evaluation of the value.

Listing 11: Execution Context

```
    Frame * frame = new Frame(context);
    Expressions * expressions = frame->lookupAs<Expressions>(sym("expr"));

    value = expressions->parse(code).value;

    return value->evaluate(frame);
```

5.4 Integers

We can look at the integer addition function for an idea of how built-in functions work.

Listing 12: Integer Addition

```
Value * Integer::sum (Frame * frame) {
```

```
    ValueT total = 0;

    // Evaluate the given arguments
    Cell * args = frame->unwrap();

    while (args != NULL) {
        // For each argument, extract it as an Integer value
        Integer * integer = args->headAs<Integer>();

        if (integer) {
            // If it was an integer, add its value to the total
            total += integer->value();
        } else {
            // If it wasn't an integer, throw an exception.
            throw Exception("Invalid Integer Value", frame);
        }

        // Move to the next argument.
        args = args->tailAs<Cell>();
    }

    // Return a new integer with the calculated sum.
    return new Integer(total);
}
```

5.5 Compilation

When using the compiler, we invoke LLVM to provide low level intrinsics. This is constructed in the same way as expressions are evaluated, but the end result is a compiled value.

Listing 13: Integer Addition

```
virtual llvm::Value * compile (Frame * frame) {
    // Find the current compiler.
    Compiler * c = frame->lookupAs<Compiler>(sym("compiler"));

    // Extract the arguments to the operator.
    Value * lhs, * rhs;
    frame->extract()(lhs)(rhs);

    // Return the LLVM remainder instruction with the two arguments.
    llvm::Value * result = c->builder()->CreateURem(lhs->compile(frame), rhs->compile(
        frame));

    return result;
}
```

6 Source Code Overview

Approximately 4100+ lines of code.

- **Kai.h** – Timer for benchmarking.
- **Token.h** – Low level parser functions.
- **Parser.h** – High level parser functions.
- **Strings.h** – String conversion functions (escape and unescape).
- **Expressions.h** – High level parser interface.
- **Frame.h** – Stack frame implementation, including tracer.
- **Function.h** – Built-in function implementation.
- **Value.h** – Value, Cell, String, Integer, Symbol, Table implementation.
- **Compiler.h** – Compiler implementation.
- **Ensure.h** – Assertions and exception handling.
- **Exception.h** – Value derived Exception class for interpreter exceptions.
- **SourceCode.h** – Line by line source code representation.
- **Terminal.h** – Interacting with terminal during interactive mode.
- **BasicEditor.h** – Command line editing for interactive mode.
- **main.cpp** – Interpreter context definition an execution.