Gnarly JIT Compiler

Spencer Tipping

July 17, 2010

Contents

1 Gnarly Bootstrap Layer

1.1 Language Definition

Gnarly is a Lisp variant that supports first-class macros. It generalizes both functions and macros by using a nonevaluative β conversion that is itself a first-class β-expander constructor. This can be combined with an evaluative identity function (written as `:`, which is also used to terminate lists) to produce functions in the normal sense.

1.1.1 Reader

Like Lisp, Gnarly provides a monolithic reader. Unlike later Lisps, however, the reader is not customizable; it follows a very simple syntax that simply constructs consed lists from its text input. The reader can be invoked from inside Gnarly as well.

The reader recognizes these forms (roughly in EBNF with regexps):

```
expression ::= <symbol> | <list>
symbol     ::= [^()\s]+
list       ::= (<expression> [<expression> ...])
```

For example, here are some expression parse trees:

```
(foo bar)   =>      (cons (cons (symbol :) (symbol foo)) (symbol bar))
(foo)       =>      (cons (symbol :) (symbol foo))
((foo))     =>      (cons (symbol :) (cons (symbol :) (symbol foo)))
bar         =>      (symbol bar)
5.0         =>      (symbol 5.0)
:           =>      (symbol :)
()          =>      (symbol :)
```

Consing is left-associative during reading for reasons explained in section 1.3.1. This has a couple of implications:

1. The head of a well-formed list is either `:` or another list
2. The tail of a well-formed list is the last element in the list

It also means that traditional list-manipulation algorithms such as `map` and `filter` traverse lists from right to left.

1.1.2 List encoding

While the original Gnarly implementation had a native cons cell, there is no such construct in this version. Instead, the reader produces curried cons-functions that are catamorphisms on a tree structure. That is, an expression produced by `read` will satisfy the following (which is an exhaustive definition):

$$(\text{cons } x\ y)\ t\ a = t\ x\ y$$
$$(\text{symbol } s)\ t\ a = a\ s$$

1.1.3 Evaluative semantics

From a logical perspective, evaluation follows these rules:

$$E[(\text{cons } x\ y)] = E[x]y$$
$$E[(\text{symbol } s)] = \text{dereference } s$$

Dereferencing of symbols is done through the global symbol table.[1] The Gnarly runtime contains symbol bindings required to change existing symbol definitions, and it contains a pair of functions to evaluate cons cells and dereference symbols.

[1] Symbols are internally unified and are treated as pointers. You can convert symbols to strings and vice versa, but the string indirection is not used for symbol table lookups.

1.2 Binary Interface

Gnarly programs have the ability to execute arbitrary machine code, and to make calls into the Gnarly runtime library using the function pointers in the global symbol table. This section defines functions and macros that are useful for exposing C++ functions to Gnarly.

Listing 1 bootstrap/core/definitions.hh

```
#ifndef CORE_DEFINITIONS_HH
#define CORE_DEFINITIONS_HH

#include "bootstrap/core/base-types.hh"
#include "bootstrap/core/namespace-macros.hh"

#endif
```

Listing 2 bootstrap/core/namespace-macros.hh

```
#ifndef CORE_NAMESPACE_MACROS_HH
#define CORE_NAMESPACE_MACROS_HH

#define BEGIN_NAMESPACE() namespace gnarly {
#define END_NAMESPACE()   }

#endif
```

1.2.1 Pointer encoding

Because all values in Gnarly can be called, they are encoded as function pointers. In a local evaluation context they can be encoded differently, but only for optimization purposes; for all observational purposes they must remain function pointers. Each function pointer refers to a unary function from values to values; thus, the type of values is recursively defined as:

$$value :: value* \to value*$$

Because many of these values are closures, I'm using classes with pure-virtual methods. I know, I know – it's slow, right? It is, but this will all be replaced by the JIT machinery once the invocation-side specializer is written.

Listing 3 bootstrap/core/base-types.hh

```
#ifndef CORE_TYPEDEFS_HH
#define CORE_TYPEDEFS_HH

#include <boost/shared_ptr.hpp>

#include "bootstrap/core/namespace-macros.hh"
```

```
7
8  BEGIN_NAMESPACE()
9
10 using boost::shared_ptr;
11
12 class gnarly_value;
13
14 typedef shared_ptr<gnarly_value> gnarly_ref;
15
16 class gnarly_value {
17 public:
18   // No cv-modifier here. The call must be able to alter the closure state, which
19   // is presumably encoded as instance data.
20   virtual gnarly_ref operator() (const gnarly_ref v) = 0;
21 };
22
23 END_NAMESPACE()
24
25 #endif
```

1.3 Runtime Bindings

Gnarly needs some symbol bindings to bootstrap the JIT compiler. These are C++ function pointers that get executed directly.

1.3.1 Evaluation

Because of the type erasure assumed in the reader's list encoding, evaluation is separated into two functions. The outer function, called `::`, invokes its parameter on the two evaluation cases. The terminology in the formal definitions is that :: $*$ evaluates cons-cells, and :: \$ evaluates symbols:

$$:: \ x = x \ (::*) \ (::\$) \tag{1}$$

$$(::*) \ x \ y = \ :: \ x \ y \tag{2}$$

$$(::\$) \ s = \text{dereference } s \tag{3}$$

Equation 2 can be optimized by η-reduction, yielding the equivalence:

$$(::*) = \ ::$$

So in fact, the definition of the evaluation functions is simply:

$$:: \ x = x \ (::) \ (::\$)$$

$$(::\$) \ s = \text{dereference } s$$

Listing 4 bootstrap/evaluation.hh

```
#ifndef CORE_EVALUATION_HH
#define CORE_EVALUATION_HH

#include "bootstrap/core/definitions.hh"

BEGIN_NAMESPACE()

class $colon$colon$dollar;

class $colon$colon : public gnarly_value {
public:
  $colon$colon () {}
  virtual ~$colon$colon () {}

  shared_ptr<gnarly_value> operator() (const shared_ptr<gnarly_value> v) const {
    return *((*v) (this)) ($colon$colon$dollar::singleton);
  }
};

static const shared_ptr<const $colon$colon>
  $colon$colon::singleton (new $colon$colon ());

class $colon$colon$dollar : public gnarly_value {
public:
  $colon$colon$dollar () {}
  virtual ~$colon$colon$dollar () {}

  shared_ptr<gnarly_value> operator() (const shared_ptr<gnarly_value> v) const {
    return reinterpret_cast<gnarly_value*> (*v);
  }
};

static const shared_ptr<const $colon$colon$dollar>
  $colon$colon$dollar::singleton (new $colon$colon$dollar ());

END_NAMESPACE()

#endif
```

1.4 Reader

The reader reads a string expression and produces an encoded syntax tree. The specifications of the syntax tree are covered in section 1.1.1.

1.4.1 Cons cell encoding

Cons cells are encoded using the binary function format discussed in section 1.1.1. As tokens are read, they are linked together to form a tree.

Listing 5 bootstrap/reader/cons.hh

```
#ifndef BOOTSTRAP_READER_CONS_HH
#define BOOTSTRAP_READER_CONS_HH

#include "bootstrap/core/definitions.hh"

BEGIN_NAMESPACE()

class reader_cons : public gnarly_value {
  class reader_cons_1 : public gnarly_value {
    const gnarly_ref t_function;

  public:
    reader_cons_1 (const gnarly_ref _t_function) : t_function (_t_function) {}
    virtual ~reader_cons_1 () {}

    gnarly_ref operator() (const gnarly_ref a_function) const {
      return *(*t_function (head)) (tail);
    }
  };

  const gnarly_ref head;
  const gnarly_ref tail;

public:
  reader_cons (const gnarly_ref _head, const gnarly_ref _tail) :
    head (_head), tail (_tail) {}
  virtual ~reader_cons () {}

  gnarly_ref operator() (const gnarly_ref t_function) const {
    return gnarly_ref (new reader_cons_1 (t_function));
  }
};

END_NAMESPACE()

#endif
```

1.4.2 Symbol encoding

Symbols are a bit simpler than cons cells. The main reason is that there is no closure state when they are singly applied (unlike cons cells, which must store a reference to `t_function`).

Listing 6 bootstrap/reader/symbol.hh

```
#ifndef BOOTSTRAP_READER_SYMBOL_HH
#define BOOTSTRAP_READER_SYMBOL_HH

#include "bootstrap/core/definitions.hh"

BEGIN_NAMESPACE()

class reader_symbol : public gnarly_value {
  class reader_symbol_1 : public gnarly_value {
  public:
    reader_symbol_1 () {}
    virtual ~reader_symbol_1 () {}

    gnarly_ref operator() (const gnarly_ref a_function) const {
      return *a_function (symbol);
    }
  };

  const gnarly_ref symbol;
  const gnarly_ref aux (new reader_symbol_1 ());

public:
  reader_symbol (const gnarly_ref _symbol) : symbol (_symbol) {}
  virtual ~reader_symbol () {}

  gnarly_ref operator() (const gnarly_ref t_function) const {
    return aux;
  }
};

END_NAMESPACE()

#endif
```

1.4.3 Parsing and tree construction

Gnarly's syntax is simple enough to just use an `in >> c` parser; that is, read a character and decide what to do with it. There is no lookahead, and recursive descent is relatively trivial. Also, because consing is left-associative, there is no

temporary allocation. Each new token in a stream can serve as a wrapper for the current value. The read function is written as *<< in Gnarly.

Listing 7 bootstrap/reader/parser.hh

```
#ifndef BOOTSTRAP_READER_PARSER_HH
#define BOOTSTRAP_READER_PARSER_HH

#include <string>
#include <istream>

#include "bootstrap/core/definitions.hh"

BEGIN_NAMESPACE()

namespace internal {

using namespace std;

static gnarly_ref read (istream &in) {
  char       c;
  gnarly_ref value;




  while (in >> c && c != ')')
}

}

class $asterisk$lt$lt : public gnarly_value {
public:
  $asterisk$lt$lt () {}
  virtual ~$asterisk$lt$lt () {}


};

END_NAMESPACE()

#endif
```