Gnarly JIT Compiler

Spencer Tipping

July 16, 2010

Contents

1 Gnarly Language Definition **1**
- 1.1 Reader . . . 1
- 1.2 List encoding . . . 2
- 1.3 Evaluative semantics . . . 2

2 Gnarly Binary Interface **3**
- 2.1 Pointer encoding . . . 3
- 2.2 Defining and calling C++ functions . . . 4

3 Gnarly Runtime Bindings **4**
- 3.1 Evaluation . . . 4

1 Gnarly 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 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 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.

1.2 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):

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

1.3 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, which is immutable.[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.

2 Gnarly 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 core/definitions.hh

```
#ifndef CORE_DEFINITIONS_HH
#define CORE_DEFINITIONS_HH

#include "core/typedefs.hh"
#include "core/namespace-macros.hh"
#include "core/ffi-macros.hh"

#endif
```

Listing 2 core/namespace-macros.hh

```
#ifndef CORE_NAMESPACE_MACROS_HH
#define CORE_NAMESPACE_MACROS_HH

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

#endif
```

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:

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

To avoid recursion in the type checker, I'm encoding these values as `void*` in C++ and casting them to function pointers at the last minute.

Listing 3 core/typedefs.hh

```
#ifndef CORE_TYPEDEFS_HH
#define CORE_TYPEDEFS_HH

typedef void *value;
typedef value* (value_fn) (const value*);
```

```
6
7 #endif
```

2.2 Defining and calling C++ functions

Internally, C++ Gnarly functions are represented by ordinary unary static functions. The JIT environment defines a scheme for currying specialized instances of generated functions, but the C++ interface doesn't presently support this.

Listing 4 core/ffi-macros.hh

```
1  #ifndef CORE_FFI_MACROS_HH
2  #define CORE_FFI_MACROS_HH
3
4  #include "core/typedefs.hh"
5
6  #define gnarly_define(name, parameter) \
7    static value* name (const value *parameter)
8
9  #define gnarly_forward_define(name, parameter) \
10   gnarly_define(name, parameter);
11
12 #define gnarly_call(fn, v) \
13   ((* static_cast<value_fn*>(fn)) (v))
14
15 #endif
```

3 Gnarly Runtime Bindings

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

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 5 core/evaluation.hh

```
#ifndef CORE_EVALUATION_HH
#define CORE_EVALUATION_HH

#include "core/definitions.hh"

BEGIN_NAMESPACE()

gnarly_forward_define(eval)
gnarly_forward_define(eval_symbol)

gnarly_define(eval, form) {
  return gnarly_call(gnarly_call(form, &eval), &eval_symbol);
}

gnarly_define(eval_symbol, symbol) {
  return *symbol;
}

END_NAMESPACE()

#endif
```