Protocol Genome Project: Functional Genes (draft)

$Id: genes.ms,v 1.2 2005/04/25 12:32:12 aki Exp $
http://www.ee.oulu.fi/research/ouspg/protos/genome/
ouspg@ee.oulu.fi

ABSTRACT

This paper presents a small domain specific language for describing structure. It is used in the Protocol Genome Project as a way to describe shared structural components of communication protocols. The language and its presented implementation effectively amount to a parser generator, though the language can also express structures that are hard to capture using grammars alone. The presented implementation is embedded into the algorithmic language Scheme.

1. Introduction

The Protocol Genome Project aims to process protocols in terms of their shared structural components - their genes. To be able to easily express and process these structures we have developed a small language for describing them. This document describes a version of that language, and a way to implement it. The language is intended to be simple, expressive enough to handle common structures from communication protocols and easy to process symbolically.

Much of the structure in protocols is syntactic and can be defined using formal grammars. In the tasks where grammars are used there is usually a strict separation between syntax and semantics. Some common structures in communication protocols, like length-payload pairs, can be viewed as leaks from semantics to syntax. They are usually simple to implement efficiently with a general purpose programming language, but require rather horrible purely grammatical constructions.

Conceptually the structures we need to express are simple. Therefore, instead of adding ad-hoc extensions to a grammar based system, or using some existing universal structure description language, we decided to make a domain specific language for the task. The language is embedded to a general purpose language. Two underlying formalisms, logic- and lambda calculus based, were considered. The language described in this document is based on ideas from both approaches.

The functional genes language consists of two parts; gene expressions and their matching functions. These correspond to regular expressions and their associated automata. The language of gene expressions is here defined with a grammar. The language consists of one or more primitive genes, and means of combining them to more complex ones. Both aggregate and primitive genes share the same structure and can be combined to make more complex genes. This allows one to quickly and safely build descriptions of complex structures from their components.

Each gene expression is a first class value in the underlying language. This means that they can be easily constructed and modified in a program. The implementation of the language described below consists of some macros and functions that translate these gene expressions to functions with a certain form. These functions to gene expressions what finite automata are to regular expressions.

The gene expressions can also be compiled to programs that perform others tasks. One implementation we have compiles them to functions that operate on suffix arrays, and one we will implement in a few months will compile them to functions that generate data instead of parsing it.

Scheme was a natural choice for implementation language. It has a hygienic macro system which makes building safe embedded languages simple. Functional programming style is also well supported, since Scheme is effectively typed Lambda-I calculus.

Some knowledge of Scheme is assumed in the following chapters.

2. The language

Below is the grammar for the gene expressions. The only primitive gene is here *bit*, which means an individual bit of data. *Union*, *catenate* and *sequence* are combining operations that can be used to construct compound genes.

```
<gene> ::= bit
     | (union <gene>+)
     | (catenate <gene>+)
     | (sequence <sequence-form>*
        (return <expression>))

<sequence-form> ::= (skip <gene>)
     | (gene <variable> <gene>)
     | (assert <expression>)
     | (let <variable> <gene>)

<expression> ::= A Scheme expression
<variable> ::= A Scheme variable
```

The convention is that a gene expression is either the name of a primitive or a previously defined gene, or an expression in parenthesis where the first element is an operation and the rest are arguments to that operation.

Each gene expression both defines a structure and gives some interpretation of it. We say that the gene evaluates to that value, given some input. The primitive gene bit evaluates either true or false depending on the input.

Union and catenate work as in formal languages. Union creates a gene that matches successfully if any of its parameter genes match. Catenate constructs a gene that matches successfully if all of the parameter genes match in that order. Union evaluates to the value of the matching gene. Catenate evaluates to a list of the values of the genes.

Sequence behaves like catenate, but instead of implicitly collecting the values returned by genes to a list, each of them is explicitly skipped with *skip* or bound to a variable with *let*

The bindings made in a sequence become lexical bindings that are visible in the subsequent genes. Each *expression* in the grammar can be any Scheme expression and can use values bound by *let* as normal variables. *Assert* checks that the given expression does not evaluate to false. This can be used for example to check that a gene has evaluated to a desired value.

Using this core language additional genes and combining operations can be defined. Below are some examples.

```
(define (epsilon value)
  (sequence
    (return value)))

(define (kleene* thing)
  (sequence
    (gene self
      (union
        (sequence
          (let this thing)
          (let rest self)
          (return (cons this rest)))
        (epsilon null)))
    (let value self)
    (return value)))

(define (kleene+ gene)
  (sequence
    (let this gene)
    (let rest (kleene* gene))
    (return (cons this rest))))
```

These behave as in regular expressions. Epsilon is a gene that always succeeds with the given value without consuming any data. The kleene operations mean zero or more and one or more repetitions of the given gene. Values are collected to a list. Kleene* could have used itself directly, but a local recursive gene was defined as an example.

```
(define (repeat count gene)
  (union
    (sequence
      (assert (= count 0))
      (return null))
    (sequence
      (let this gene)
      (let rest (repeat (- count 1) gene))
      (return (cons this rest)))))

(define (until step terminal)
  (union
    (sequence
      (let match terminal)
      (return (list match)))
    (sequence
      (let this step)
      (let rest (until step terminal))
      (return (cons this rest)))))
```

Repeat means a repetition of a given gene a given number of times. This is a common construct in communication protocols. Note that the count may be a variable bound in a sequence and can therefore depend on the preceding input. Until means a sequence of zero or more given step genes followed by a given terminal gene. The values are in both cases collected to a list.

```
(define (integer size)
  (sequence
    (let bits (repeat size bit))
    (return (bits->integer bits))))

(define byte (integer 8))

(define (literal wanted gene)
  (sequence
    (let value gene)
    (assert (equal? value wanted))
    (return value)))
```

Integer interprets a fixed length bit sequence as an unsigned integer. Canonical bit order is assumed unless otherwise stated. Literal is a gene that evaluates to a given value.

3. An implementation

The basic idea of the implementation is to represent all genes as functions written in explicit success and failure continuation passing style. Lexical bindings are used to store all state information. This technique is commonly used in compiling logic based languages to functional languages. Gene expressions are directly compiled to functions by defining the primitive combining operations as macros, and the primitive gene as a function. The target language is effectively Lambda-I calculus, which makes reasoning about genes and their behavior relatively simple.

We use bit as the primitive gene. Input data is represented as a list of boolean values. The only case in which the initial gene gene will fail is if there is no more input data. In that case the failure continuation *ft* is invoked. Otherwise the first bit and rest of the data are passed to the success continuation *sc*

```
(define bit
  (lambda (sc ft data)
    (if (null? data) (ft)
      (sc (car data) (cdr data)))))
```

The remaining three primitives construct new genes. Sequence is a macro that first expands to a new gene - a lambda expression with the same form as the bit - and then proceeds to process the contents. The let form binds the value returned by a gene to a given variable and skip behaves similarly but binds the value to a variable that does not occur free in rest of the expression. Assert calls failure continuation if the given expression evaluates to false. Because each gene is compiled to a function, Scheme’s local mutual recursive function definition form can be used also for internal genes.

```
(define-syntax sequence
  (syntax-rules (assert return skip let gene)
    ((sequence (a . b) . c)
      (lambda (sc ft data)
        (sequence 42 sc ft data (a . b) . c)))
    ((sequence 42 sc ft data
      (gene name1 val1) (gene name2 val2) ... . rest)
      (letrec ((name1 val1) (name2 val2) ...)
        (sequence 42 sc ft data . rest)))
    ((sequence 42 sc ft data (let var thing) . rest)
    (thing
      (lambda (var data)
        (sequence 42 sc ft data . rest))
      ft data))
  ((sequence 42 sc ft data (skip thing) . rest)
    (thing
      (lambda (fresh data)
        (sequence 42 sc ft data . rest))
      ft data))
  ((sequence 42 sc ft data (assert exp) . rest)
    (if exp (sequence 42 sc ft data . rest) (ft)))
  ((sequence 42 sc ft data (return value))
    (sc value data))))
```

Union constructs a new gene, where the success continuation of each sub-gene is connected to the new one, and each failure continuation to the remaining sub-genes or the new failure continuation. Catenate simply collects values from each gene and constructs a list of them.

```
(define-syntax union
  (syntax-rules ()
    ((union 42 sc ft data a)
      (a sc ft data))
    ((union 42 sc ft data a . b)
      (a sc
        (lambda () (union 42 sc ft data . b))
        data))
    ((union 42 sc ft data) (ft))
    ((union . stuff)
      (lambda (sc ft data)
        (union 42 sc ft data . stuff)))))

(define-syntax catenate
  (syntax-rules ()
    ((catenate a . b)
      (sequence
        (let this a)
        (let rest (catenate . b))
        (return (cons this rest))))
    ((catenate)
      (epsilon null))))
```

These four definitions are all it takes to compile gene-expressions into executable functions in Scheme. The small core is useful, because the system can easily be modified and re-targeted. A number of variations of the primitive operations have been tested. Each operation also has simple properties, which makes proving assertions about them relatively easy.

There are many alternatives to this implementation. In this version the first gene of a union that matches will be the value of the union. The suffix array based version processes each of the possibilities.

Scheme provides first class continuations which can be used for backtracking. We initially thought that real continuation could be useful for adding another backtracking layer when processing suffix trees or arrays, but it turned out that the one provided by genes is sufficient with only a few modifications. In another languages genes can be for example defined as networks of objects or as code for a stack based machine.

4. Case studies

We have written some tests to see how different versions of the language behave. Currently the largest gene has a description of about 250 lines. It defines the parser for of the Scheme implementation used for prototyping purposes. Apart from having byte as the primitive piece of data, it uses the same macros and derived operations that are defined in this paper.

A protocol related test is filtering IPv4 packets from a packet capture file. The packet structure is described with a gene of about 20 lines and returns the payload of the packet.

A simple calculator program was written to demonstrate how genes can be used to implement small languages. It calculates values of fully parenthesized arithmetic expressions and has a description of some 50 lines.

5. Future applications

The language is still evolving. Efficiency and portability issues will be addressed if the language proves to be useful.

One obvious sample application would be to write an easily extensible and traffic analyzer. The simplest approach would be to write a large number of packet descriptions and a matcher that analyzes for example a packet capture file against the given descriptions. Note that this is **not** the goal of the Protocol Genome Project. There are already good tools for this task, and we do not have enough resources to start writing a useful set of protocol descriptions by hand. We will probably write a simple traffic analyzer and descriptions for a few protocols for testing purposes. The advantage of our approach would be that protocol descriptions could probably be relatively simple and could be loaded from plain text files to the program. It might become a useful tool as such, provided we could make it sufficiently easy and extend.

The genes could also be used to write verified extended parsers. We believe that there is need for an embedded language similar to regular expressions that would allow safe and easy processing of possibly malicious or mutated input data.

In Protocol Genome Project the genes are used as a structure description language. The current prototype searches for known genes from input data, constructs new ones for frequent repetitions, scores them with an evaluation function and matches them against a suffix vector. The current main goal is to use various gene expression inferring and learning techniques to build a realistic autonomous protocol reverse engineering tool.

References:

[Scheme]

R. Kelsey, W. Clinger, J. Rees (eds.), *Revised5 Report on the Algorithmic Language Scheme,* Higher-Order and Symbolic Computation, Vol. 11, No. 1, August, 1998 and ACM SIGPLAN Notices, Vol. 33, No. 9, September, 1998

[Lambda-I]

A. Church *The Calculi of Lambda Conversion. (AM-6) (Annals of Mathematics Studies)* Princeton University Press; (January 1, 1985)

Appendix A: IPv4 header

```
(sequence
  (let ip-version (integer 4))
  (let header-length (integer 4))
  (let service-type byte)
  (let total-length (integer 16))
  (let identification (integer 16))
  (skip zero-bit)
  (let DF-bit bit)
  (let MF-bit bit)
  (let fragment-offset (integer 13))
  (let time-to-live byte)
  (let protocol byte)
  (let header-checksum (integer 16))
  (let source-address word)
  (let dest-address word)
  (let options (repeat (- header-length 5) word))
  (let payload (repeat (- total-length (* header-length 4)) byte))
  (return payload))
```