Lisp in Summer Projects Submission

Submission Date	2013-10-03 21:45:13
Full Name	Marc A Coram
Country	United States
Project Name	primrec
Type of software	command-line/terminal app
General category	other
LISP dialect	Racket
GitHub URL	https://github.com/mcoram/primrec
Did you start this project?	Yes, all the code is written by me
Project Description	I want to describe my project in this form.
Purpose	To explore the space of primitive recursive functions.
Function	The code systematically evaluates primitive recursive functions of arity up to 3 of increasing complexity, keeping track of which functions are observably distinct. The results can be queried to find, e.g. that if f(0)=1, f(1)=1, f(2)=2, f(3)=6, then f(4) is "probably" 24.
Motivation	Curiosity mainly. How far could I push the idea that simple constructive objects should have simple explanations, in this case as short expressions in a syntax for primitive recursive functions. Also what functions/numbers are "objectively" simple? E.g. 22539988369407 is simple because it can be written using a mere 22 symbols: (C10 (C21 (R1 (C23 (R1 (C23 (R1 (C13 S P32) S) P32 P32) S) P32 P32) S) S S) (C10 S 0))
Audience	Anyone else curious about making a constructive basis for inductive inference.
Methodology	To explore the space of primitive recursive functions effectively one must evaluate millions of functions, but not

get bogged down by the plethora of ones that are surprisingly slow (albeit terminating). Neither do you want to get lost among the mire of functions that are operationally equivalent to functions you've already identified: you'd already be exploring all the possible compositions of the previously identified function and you don't want to replicate that effort ad nauseam. As a practical expedient, functions are distinguished by evaluating them only on a panel of small inputs (e.g. 0..24 for arity 1 functions). The execution is in a thread that's only allowed 1/4 second to execute. Different strategies for dealing with functions that fail to execute in this time, and therefore might be interestingly different, though we don't know, were considered (see the git branches).

The core of the algorithm uses a "co-recursively defined" vector containing 4 lazy-vectors. Entry i of the vector is the lazy-vector holding results of arity i (i=0..3). The j'th entry of the i'th lazy-vector is itself a list of the distinct primitive recursive functions of arity i which can be written with exactly j symbols. The lazy-vectors are implemented by a (reusable perhaps) Racket module that is more-or-less a memoizer for inductively defined functions, "extenders", of 0,1,2,... (if you look up the value at 10 it is designed to ensure the values of the extender at 0..9 are known first).

To give the gist of how the extender for the arity 1 case is defined, for example, consider evaluating the extender at 10. The objective is to identify all arity 1 functions with exactly 10 symbols that can be constructed out of our (pre-screened for novelty) smaller functions. The new construct could begin with R0, representing primitive recursion, which needs an arity 2 function with, say, 6 symbols (look up all the pre-computed possibilities) and an arity 1 function with, say, 3 symbols, where 6 and 3 range over all the choices so that the total count works out (1+3+6=10). Alternatively it could be a composition of two arity 1 functions (...), or of an arity 2 with two arity 1s or an arity 3 with 3 arity 1s.

Conclusion

The software is effective for systematically exploring primitive recursive functions of arity up to 3. Various findings about the functions thus encountered are in the "notes.txt" file. For example, a number which can be written with 23 symbols as

(C10 (C21 (R1 (C23 (R1 (C13 (R0 (R1 (C13 S P32) S) 0) P32) S) P32 P32) S) S S) (C10 S 0))

is much larger than the result of the following algorithm:
set x=1541
3080 times: set x=2^x
return x
which is just absurdly large.

A simple web interface is provided to query arity 1 functions.

Limitations: Only uses 1 core. Handles slow functions in a limited way. Results are sensitive to the particular definitions of the primitive recursive symbols.

Build Instructions

Batteries included; just make sure Racket is installed and git clone it.

Test Instructions	I do not have automated tests. There are some (manual) tests in the lazy-vector.rkt file, or sprinkled throughout.
Execution Instructions	Try the web version. From the command line use: racket -t web.rkt Then browse to http://localhost:8000/ To try predict-extension from the command line use: racket -t predict-extension.rkt -m Or load and run predict-extension.rkt in DrRacket and execute "(main)" To compute the primitive recursive function listings that it uses, run pr04 from the command line like so: racket -t pr04.rkt -m >out/functions-full.log Browse the progress on the log with: tail -f out/functions-full.log Also useful to check progress: grep extender out/functions-full.log
Screen shots	screenshot_1126.png screenshot_124.png screenshot_234.png
Official	I have read rules and have abided by them. I am 18 years of age or older. I am not living in Brazil, Quebec, Saudi Arabia, Cuba, Iran, Myanmar (Burma), North Korea, Sudan, or Syria.