Knowledge Lab

Birkbeck
UNIVERSITY OF LONDON

Research Student
Rik Howard

Supervisors
Keith Mannock
Trevor Fenner

Procedural-Functional Programming

Research Aims

The aim is to design, formalise and (prototypically) implement a general-purpose programming language and machine. The goal for the language is to provide a clean syntax for capturing programming ideas within a semantic framework about which it is relatively easy to reason. The proposal is to deliver a procedural-functional language-machine with multiple, communicating processes.

Research Methodology

The functional (declarative, deterministic) core is to be provided by an extended, impure lambda calculus. There is to be support for pure first-class functions and recursion; normal-order evaluation and referential integrity are to be exhibited.

```
coutdown n :=
    Zero n?
        0:
    countdown (decrement (n)).
```

Figure 1, a function defined (`:=`) by a boolean conditional disjunction (`?:`)

The procedural (imperative, non-deterministic) wrapping is to be provided by a simple syntactic-semantic extension to the core and the provision of assignable arguments (out-variables). The out-variables provide the procedures with a mechanism to return (non-deterministic) values.

```
echo :=
    read string!
        write string:
     error.
```

Figure 2, a procedure defined by a void-or-other-valued conditional disjunction (`!:`) and using an out-variable (`string`)

Multiple, communicating processes are to be supported by providing functionality to spawn, send messages to and receive messages from other threads.

Research Approach

The approach is to develop confidence in a minimalistic subset of the language that is prepared in such a way as to be readily extensible with secondary features and amenable to optimisation. The minimal language is to be equipped with Boolean values, natural numbers, characters, strings, pairs and lists. In-built functions are to provide core functionality such as for constructing and accessing pairs.

```
ns :=
    nns n := (n, nns (increment (n)));
    nns 0.
show ns :=
    write (concatenation (string (left ns)) " ")!
        show (right ns):
    error.
> show ns
0 1 2 3 4 5 6 ...
```

Figure 3, an execution demonstrating nested function definition, pairs and normal-order evaluation

A further value is introduced, thought of as *void* or *okay*, to indicate the successful execution of a procedure. In-built procedures are to be provided for reading from stdin and writing to stdout and for spawning and communicating with other processes.

```
receiveToWrite := recv msg! write msg: error.
sendString string :=
    spawn receiveToWrite [] pid! send pid string: error.
```

Figure 4, a program utilizing multiple, communicating processes

The syntax presented in the figures has been slightly idealized as the combined functionality is currently spread across several projects. Please see the links below.

Web

- http://www.dcs.bbk.ac.uk/~rik/gallery
- https://bitbucket.org/rik_howard

Birkbeck
UNIVERSITY OF LONDON

Department of Computer Science and Information Systems