Playing with Games: the Games Workbench

Juliusz Chroboczek

February 12, 2001

Introduction

The Games Workbench (GWB) is a Common Lisp program designed to serve as a framework for experimenting with Game Semantics.

1 Using the Games Workbench

The user interacts with the interactive GWB toplevel. At the GWB prompt,

```
GWB>
```

the user may type any GWB *command*.

1.1 GWB commands

A GWB command is either a Lisp symbol, as in `quit`, or a list the first member of which is a Lisp symbol, as in

```
(display Y-run)
```

You may obtain the full list of commands by typing `help` at the GWB toplevel.

1.1.1 Running and defining terms

The **run** command takes a single argument, which is a term to run. The syntax of terms is defined in Section 1.2.

```
GWB> (run (+ 3 4))
7
5 moves played.
```

In order to avoid having to retype a term multiple times, it is possible to define an abbreviation using the **define** command. Note that this abbreviation is defined at the source level, and is recompiled every time it is used; such abbreviation are very different from bound variables and imperative variables.

```
GWB> (define add (lambda (x y) (+ x y)))
ADD
GWB> (run (add 3 4))
7
91 moves played.
```

The **clear** command may be used to clear definitions.

1.1.2 Loading files

The **load** command executes the contents of a file as though it were typed at the GWB toplevel. The file should consist of a sequence of GWB commands.

```
GWB> (load "fact.gwb")
FACT0
FACT
```

1.1.3 Inspecting positions

There are two basic ways in which it is possible to inspect the moves played by a strategy, on-the-fly tracing and displaying remembered runs.

Tracing strategies If M is a term, then (**trace** M) behaves just like M, but has the side-effect of printing out every move that is being played by the strategy associated to M.

The **trace** form takes one or two arguments. The first is the term to trace; the second is optional, and is an arbitrary *name* that can be used to track multiple interleaved traces.

```
GWB> (run ((lambda x (+ x x)) (trace 5 five)))
FIVE O Q - ? NIL
FIVE P A - 5 1
FIVE O Q - ? NIL
FIVE P A - 5 1
10
25 moves played.
```

The columns printed out are, respectively, the name of the strategy, the player who plays this move, whether the move is a question or an answer, the component of the move, the token of the move, and its justification pointer, represented as the number of moves that the pointer spans.

Remembering runs If M is a term, then (**remember** M n) behaves just like M, but has the side-effect of remembering under the name n all the moves played by M under the name n. Remembered runs take up memory, and they should be erased using the **forget** command.

The GWB provides a number of commands that can be used to inspect remembered runs; they all take a single argument which is the name of a remembered run. The simples is **components**, which simply lists the set of components that contain moves. The command **display-simple** uses the same format as **trace** except for the first column. The command **display-tabular** which displays moves in columns corresponding to components. Finally, **display** pops up a window with a graphic representation of a position.

The graphic representation may be saved in an EPS file by using the **print-eps** command.

```
GWB> (run ((lambda x (+ x x)) (remember 5 five)))
10
25 moves played.
GWB> (display-simple five)
```

```
P Q - ? NIL
O A - 5 1
P Q - ? NIL
O A - 5 1
GWB> (display-tabular five)
?
5
?
5
GWB> (display five)
GWB> (print-eps five "five.eps")
```

1.2 Syntax of terms

The GWB implements a call-by-name programming language containing with features from PCF and Idealised Algol, as well as non-local control operators in the form of **call/cc**.

1.2.1 Functional core

The basic syntax consists of $\lambda -$ *abstraction* and function application:

$$(\textbf{lambda}\ x\ M)$$

$$(M\ N)$$

Two abbreviations are provided for convenience. The syntax

$$(\textbf{lambda}\ (x_1\ x_2 \cdots x_k)\ M)$$

is a shorthand for

$$(\textbf{lambda}\ x_1\ (\textbf{lambda}\ x_2\ (\cdots(\textbf{lambda}\ x_k\ M)\cdots)))$$

The syntax

$$(M_1\ M_2 \cdots M_k)$$

is an abbreviation for

$$(\cdots(M_1\ M_2)\cdots M_k)$$

Functional constants The ground values are **true**, **false** and integers. In addition, **omega** is used for representing looping terms, and **top** for broken terms. (They are distinct as per the theory, but are actually implemented alike.)

Higher order constants include the combinators **compose**, **S**, **K**, **I**, **Y**. The constant $=$ takes two integer arguments and returns a Boolean; the constant **if** takes one Boolean argument and two arbitrary arguments.

Finally, the constants **pair**, **p0** and **p1** implement (unlifted) products.

1.2.2 Control operators

The constant **call/cc** takes a single argument which is a function that takes the current continuation as argument. This constant works at higher order.

1.2.3 Imperative features

The GWB implements the imperative features of Idealised Algol (with active expressions). The basic construct is **new**, with the syntax

$$(\textbf{new}\ x\ n\ M)$$

where x is the name of the imperative variable, n is a litteral (compile-time) integer that specifies the initial value, and M is the body where x is visible.

An imperative variable is dereferenced using the constant **deref**, which takes a single imperative variable, and set using **set**, which takes an imperative variable and a value.

Sequencing is provided in the form of **seq**, which takes two terms, evaluates them in order, and returns the result of the second. As a convenience, the form

$$(\textbf{begin}\ M_1\ M_2 \cdots M_k)$$

is equivalent to

$$(\textbf{seq}\ M_1\ (\textbf{seq}M_2\ (\textbf{seq}\ \cdots M_k \cdots)))$$

The void imperative instruction, dear to some of my former lecturers, is provided as the constant **dummy**.

Additional forms and constants The constant **print** behaves like the identity on the integers; it has the side effect of printing out the value of its argument whenever it is being evaluated.

The forms **trace** and **remember** are described above.

1.3 Summary of syntax

1.3.1 Commands

- **quit** and **help**;
- (**load** f);
- (**run** M);
- (**define** n M), **list** and **clear**;
- (**components** n);
- (**display-simple** n), (**display-tabular** n) and (**display** n);
- (**print-eps** n f).

1.3.2 Terms

Syntax:

- lambda-abstraction: (**lambda** x M) or (**lambda** $(x_1 \cdots x_k)$ M);
- function application: $(M_1 \cdots M_k)$;
- imperative variable binding: (**new** x n M).

Constants:

- **true**, **false**, integers, **omega** and **top**;
- **compose**, **S**, **K**, **I**, **Y**, **if**;
- **pair**, **p0**, **p1**;
- **call/cc**;
- **set**, **deref**, **skip**, **seq**;
- **print**.

Macros and special forms:

- (**trace** M [n]) and (**remember** M n);
- (**begin** $M_1 \cdots M_k$).

2 Examples

2.1 Functional core

2.1.1 Recursive functions

The GWB does not provide for recursive function definitions; however, it does provide a fixpoint combinator.

```
GWB> (define fact0
       (lambda (f n)
         (if (= 0 n)
           1
           (* n (f (- n 1))))))
FACT0
GWB> (define fact
       (Y fact0))
FACT
```

This is a call-by name factorial.

```
GWB> (run (fact 1))
1
641 moves played.
GWB> (run (fact 2))
2
1387 moves played.
GWB> (run (fact 3))
6
2425 moves played.
GWB> (run (fact 4))
24
3755 moves played.
GWB> (run (fact 5))
120
5377 moves played.
```

Let's see what's going on.

```
GWB> (run ((remember Y Y-run) fact0 0))
1
187 moves played.
GWB> (display-simple Y-run)
P Q 11 ? NIL
O Q 011 ? 1
P Q 010 ? 1
O Q 10 ? 3
P A 10 0 1
O A 010 0 3
P A 011 1 5
O A 11 1 7
GWB> (display-tabular Y-run)
      ?
    ?
  ?
?
0
  0
    1
      1
GWB> (run ((remember Y Y-run) fact0 1))
1
641 moves played.
GWB> (display Y-run)
```

The results of this last command are in Fig. 1.

2.1.2 Implementing lists

A list has three methods: the **null** predicate, **car**, and **cdr**. The latter two are only defined for non-**null** lists.

```
GWB> (define cons
       (lambda (head tail)
         (pair false
               (pair head
                     (pair tail top)))))
CONS
GWB> (define nil
       (pair true top))
NIL
GWB> (define null p0)
NULL
GWB> (define car
       (lambda (x)
         (p0 (p1 x))))
CAR
GWB> (define cdr
```

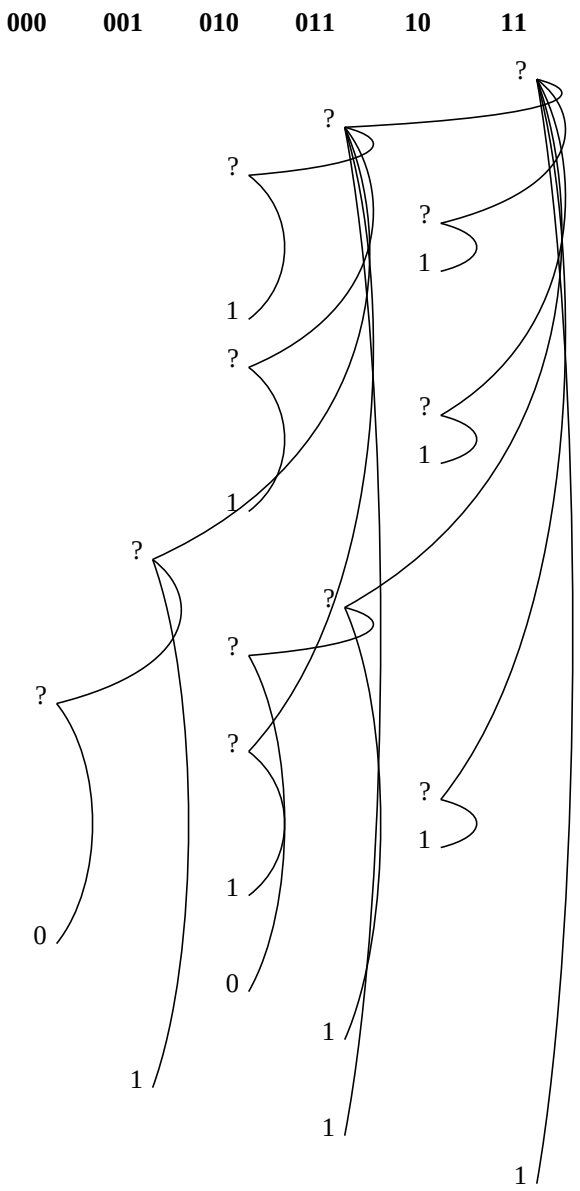

Figure 1: The Y combinator combuting the factorial in call-by-name

```
       (lambda (x)
         (p0 (p1 (p1 x)))))
CDR
```

Let's define a sample list and explore it:

```
GWB> (define l (cons 1 (cons 2 (cons 3 nil))))
L
GWB> (run (null l))
FALSE
47 moves played.
GWB> (run (car l))
1
121 moves played.
GWB> (run (car (cdr l)))
2
317 moves played.
```

A function for summing lists:

```
GWB> (define sum0
       (lambda f
         (lambda (x)
           (if (null x)
             0
             (+ (car x) (f (cdr x)))))))
SUM0
GWB> (define sum (Y sum0))
SUM
GWB> (run (sum l))
6
4213 moves played.
GWB> (run (sum (remember l list-run)))
6
4213 moves played.
GWB> (display list-run)
```

(See Fig. 2.)

```
GWB> (run (Y (remember sum0 sum0-run) l))
6
4213 moves played.
GWB> (display sum0-run)
```

(See Fig.3.)

2.2 Control operators

```
GWB> (define add-or-leave
  (lambda (n m k)
    (if (= 0 n)
      (k 0)
      (+ n m))))
```

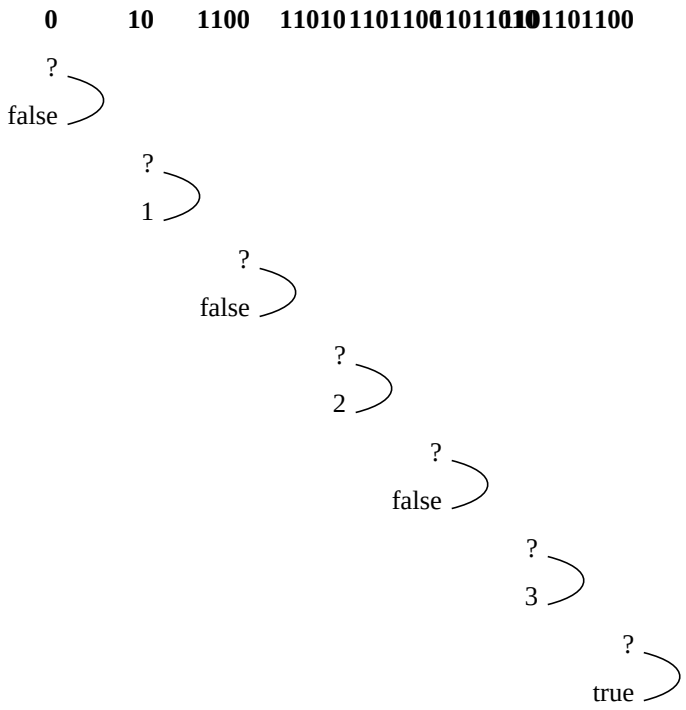

Figure 2: A list being explored

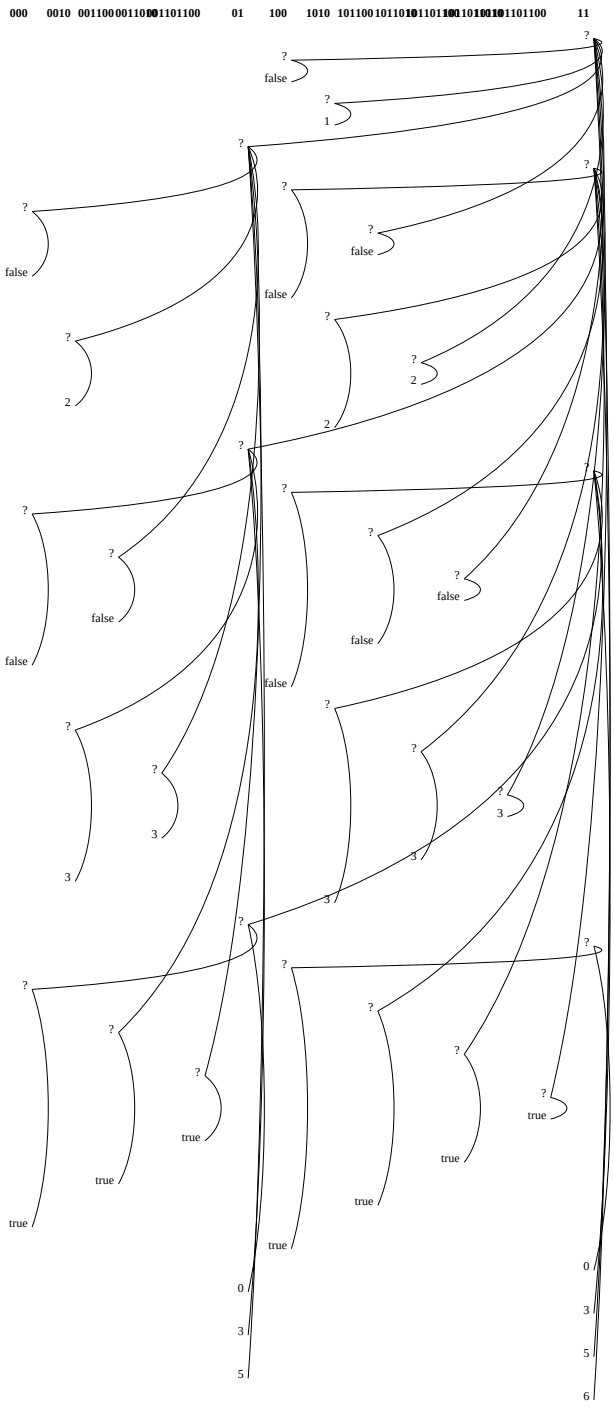

Figure 3: Summing a list

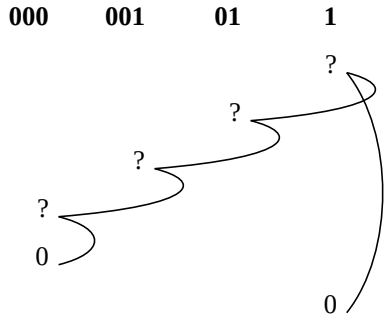

Figure 4: A position of **call/cc**

```
ADD-OR-LEAVE
GWB> (run (call/cc (lambda k (+ (add-or-leave 0 47 k) omega))))
0
935 moves played.
GWB> (run (call/cc (lambda k (add-or-leave 0 omega k))))
0
925 moves played.
GWB> (run ((remember call/cc call/cc-run) (lambda k (add-or-leave 0 omega k))))
0
925 moves played.
GWB> (display call/cc-run)
```

(See Fig. 4.)

2.3 Imperative features

2.3.1 Simple imperative example

```
GWB> (define increment
       (lambda x
         (begin
           (print (deref x))
           (set x (+ 1 (deref x))))))
INCREMENT
GWB> (define thrice
       (lambda f
         (new x 0
           (begin
             (f x)
             (f x)
             (f x)
             (deref x)))))
THRICE
GWB> (run (thrice (remember increment increment-run)))
==> 0
```

```
==> 1
==> 2
3
1289 moves played.
GWB> (display increment-run)
```

See Fig. 5.

2.3.2 Call-by-value

With imperative variables, we can force evaluation of an integer, and implement call-by-value functions.

```
GWB> (define cbv-fact0
       (lambda (f n)
         (new var 0
           (begin
             (set var n)
             (if (= 0 (deref var))
               1
               (* (deref var) (f (- (deref var) 1))))))))
CBV-FACT0
GWB> (define cbv-fact
       (Y cbv-fact0))
CBV-FACT
GWB> (run ((remember Y Y-cbv-run) cbv-fact0 1))
1
5831 moves played.
```

See Fig. 6 and compare with Fig. 1.

Acknowledgements

The GWB was initially written while I was working on my Ph.D. at the University of Edinburgh under the supervision of Samson Abramsky, who encouraged me to explore the possibility to implement Game Semantics. I would also like to acknowledge the encouragement I received from Kohei Honda.

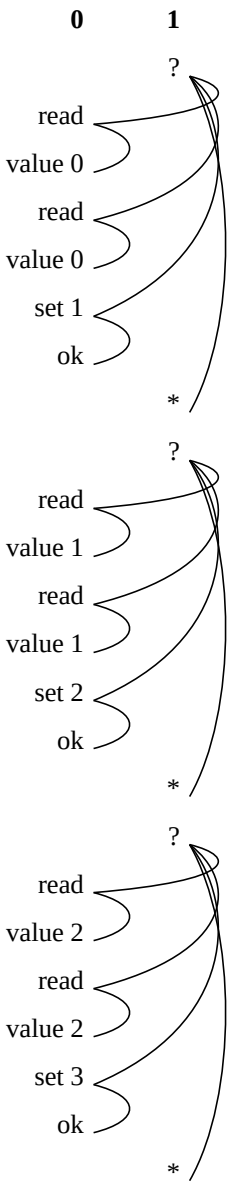

Figure 5: A position of the function **increment**

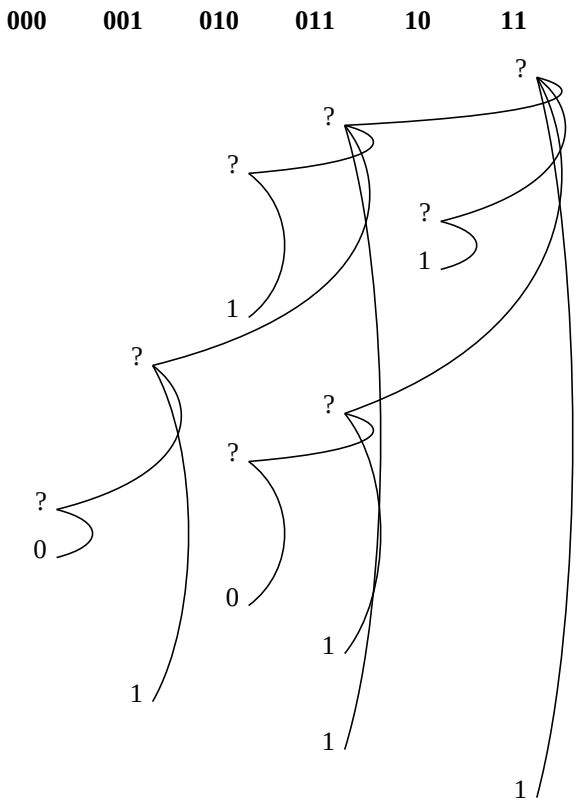

Figure 6: The Y combinator combuting the factorial in call-by-value