Gamboling with Gambit: Applications and Libraries

Bradley Lucier

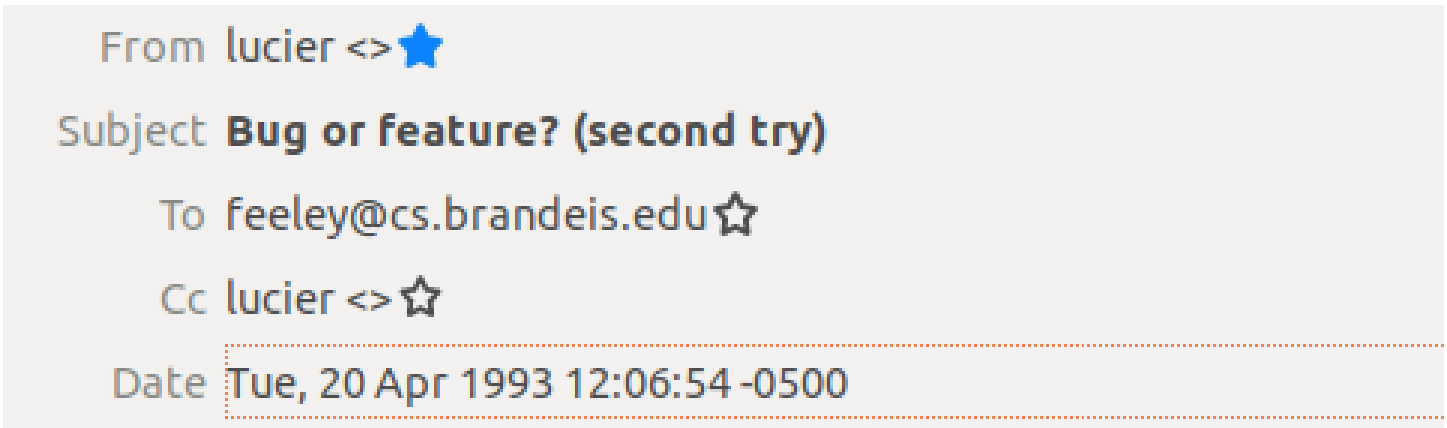
Gambiteer on Github

DEFINITION

To Gambol:

1. To skip about, as in dancing or playing; frolic.

FIRST CONTACT


From lucier <>
Subject Bug or feature? (second try)
To feeley@cs.brandeis.edu
Cc lucier <>
Date Tue, 20 Apr 1993 12:06:54 -0500

FIRST RESPONSE


Feature!

APPLICATIONS

Genetic Programming System, 1996–1998

Evolve programs to solve image-processing problems.

Each program was run on each pixel (256K pixels/image); hundreds of programs per generation.

I redid the representation in Gambit's compiler of sets of variables/labels/$\cdots$; tripled the speed of the compiler.

Used knowledge of Gambit Virtual Machine (GVM) registers, ended up doing "lambda-lifting" (converting a procedure's free variables to explicit arguments) by hand.

Result: Fastest genetic programming system in existence—faster than C, faster than assembler—with an overhead of 11 machine cycles per pixel.

Numerical Partial Differential Equations, 1999–2016

An functional/object-oriented system for solving elliptic, parabolic, and hyperbolic PDEs using the finite element method in two space dimensions.

Used as a pedagogical tool to teach upper-level graduate students, who had to write code to add to the library to solve a project problem.

Most of time was taken up in sparse matrix–vector multiplication, which was limited in speed by the time for memory access.

Sparse matrix–vector multiplication **ran as fast** in Gambit as in C.

Homework on the Web System, 2000-2009

Only system that accepted all correct answers and rejected all incorrect answers.

Was very careful that 0.1 was translated into $1/10$ and not the floating-point number closest to $1/10$.

Used five Maple processes for backend symbolic processing.

Was used by 600 students each fall semester, 800 each spring.

Used text files of S-expressions and RCS (the original "Revision Control System") to store student records.

Ran interpreted on an Ultrasparc system, satisfied 40,000 transactions/hour.

LIBRARIES

Meroon

An object system written by Christian Queinnec.

A very sweet design point: Single inheritance; generic functions with methods; multiple-argument method dispatch, with single-argument dispatch as a very fast special case.

Meroon is written in Meroon (with a bootstrap process), with a compile time MOP (Meta-Object Protocol) that allows you to modify the behavior of Meroon at compile time. (The Common Lisp Object System (CLOS) has a run time MOP.)

Very closely integrated into Gambit. Used extensively in my Numerical PDE application.

BIGNUMS!

Bignums are integers of arbitrary size.

In June, 1998, Marc asked me if I'd like a project: To rewrite Gambit's bignum library based on the 1986 paper:

Reconfigurable, Retargetable Bignums:

A Case Study in Efficient, Portable Lisp System Building

Jon L White

Lucid, Inc.
707 Laurel Street
Menlo Park, CA 94025

JonL White's big idea

To introduce a **small number of primitives** that can be written in C, Assembly, Java, Python, etc., and write the rest of the system in Scheme.

Access the bits of bignums in chunks appropriate to the operation, perhaps 64 bits for addition/subtraction, 32-bits for multiplication, even smaller for specialized operations. Universal back end accesses bignums in 14-bit chunks!

Maximum portability—32 and 64 bit machines, big and little-endian, with or without C's `long long`, etc.

Writing library in Scheme $\Longrightarrow$ can try **more sophisticated algorithms**.

Arithmetic between exact and inexact numbers

Beginning around 1998, I started to use **Nonstandard Analysis** as a heuristic for interpreting arithmetic in Scheme.

Nonstandard real numbers can be interpreted as certain **sequences of real numbers**.

Each **standard real** number is interpreted as a **constant sequence**, e.g.,

$$0 \to (0, 0, 0, \dots), \ 1 \to (1, 1, 1, \dots), \ \text{etc.}$$

Operations $(+, \times, \text{etc.})$ are **componentwise**.

Nonstandard real numbers

But there is also

$$\epsilon = (1, \frac{1}{2}, \frac{1}{3}, \dots),$$

which is greater than $0 = (0, 0, \dots)$, but less than any positive standard real number.

And

$$\epsilon^{-1} = (1, 2, 3, \dots),$$

which is bigger than any standard real number.

Nonstandard real numbers include **infinitesimals** and **infinities**.

Modeling floating-point numbers

Floating-point numbers include `+inf.0` and `-inf.0`, which we'll model with ϵ^{-1} and $-\epsilon^{-1}$.

They also include `+0.` and `-0.`, modeled with ϵ and $-\epsilon$.

And `+1.0` is modeled by any sequence converging to 1.

But we could choose **other number sequences** to model `+0.` and `+inf.0`.

So we know, e.g., `(* +0. -0.)`$\Rightarrow$`-0.`, but `(* +0. +inf.0)` **is indeterminate**, it could be anything, so in floating-point arithmetic it returns `+nan.0`, a **Not-a-Number**, a floating-point "number" that is not associated with any real number.

What is `(* 0 +inf.0)`?

Previous programming systems didn't ask what happens when you mix **exact** (`0`) with **inexact** (`+inf.0`) numbers, they just **converted** the exact numbers to floating-point and carried on.

We'll model **exact** Scheme numbers (`0`, `3/4`, etc.) as **standard real numbers** and **floating-point** numbers (`-0.`, `+inf.0`, `2.0`, etc.) as **nonstandard reals**.

So **we interpret** `(* 0 +inf.0)` as

$$0 \times \epsilon^{-1} = (0, 0, 0, \dots) \times (1, 2, 3, \dots) = (0, 0, 0, \dots) = 0$$

or **exact zero**. Similar examples abound.

Has effects throughout the runtime library, especially to simplify the routines for complex numbers.

LAST POINT

THANK YOU MARC!

References

Genetic programming:

https://www.math.purdue.edu/~lucier/692/gp98.pdf

Numerical Partial Differential Equations:

https://www.math.purdue.edu/~lucier/pde-papers/pde-in-scheme.pdf

Homework on the web:

https://www.math.purdue.edu/~lucier/SAGE.pdf