Goo Implementation

Jonathan Bachrach
MIT AI Lab

Outline

- Goals
- AST
- Runtime
- Compilation to C
  - AST transformations
  - C output strategy
  - C runtime
- Bootstrapping
- Beyond

- Based on
  - LiSP chaps 6, 9 and 10
  - Fun-o-dylan

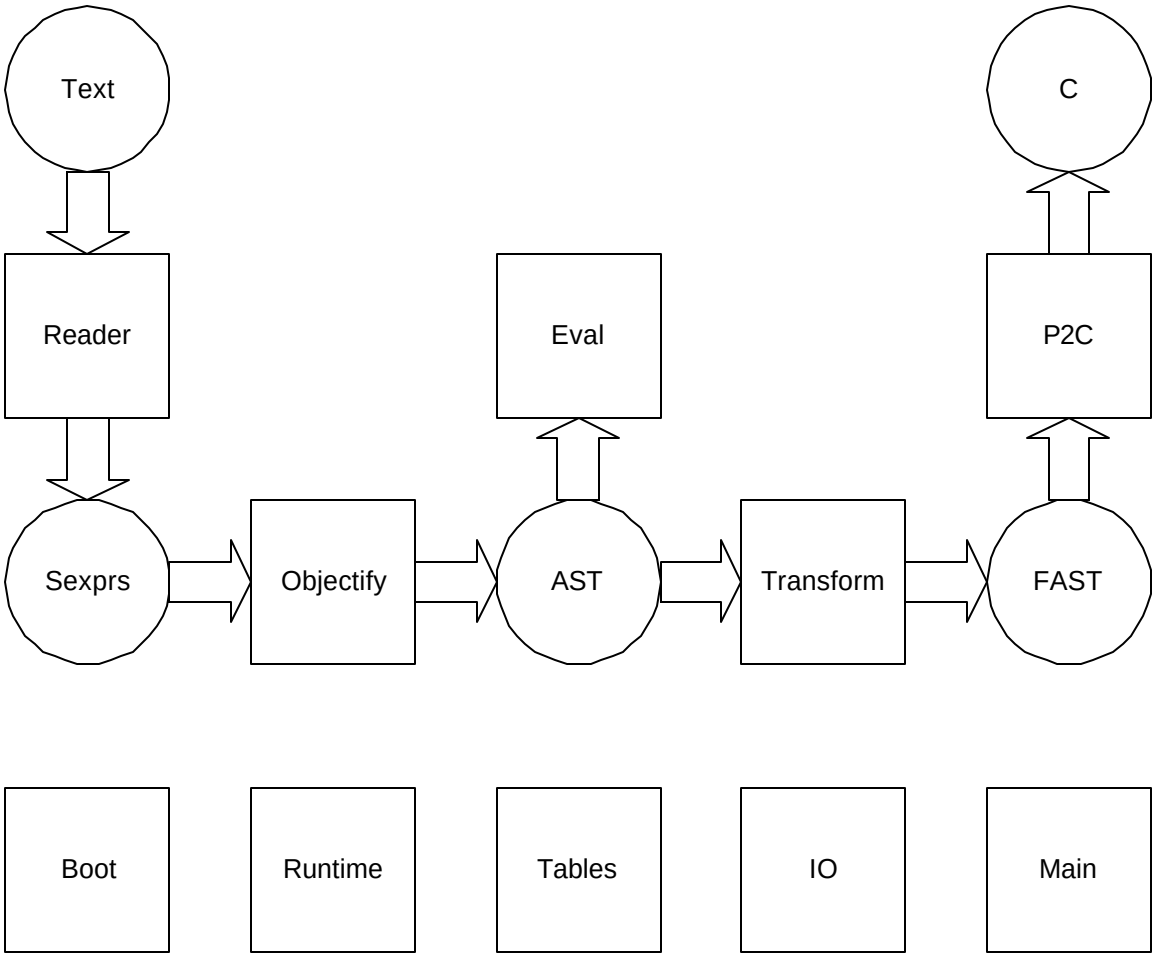
Goo Goals

- Minimal complexity of combined
  - Language
  - Runtime
  - Compiler

What Goo’s Not

- High performance
- Sophisticated

Goo Architecture



Abstract Syntax Tree

- Object-oriented syntactic tree representation
- Program converted into an object
- Syntax normalized and checked
- AST objects can easily be evaluated and transformed

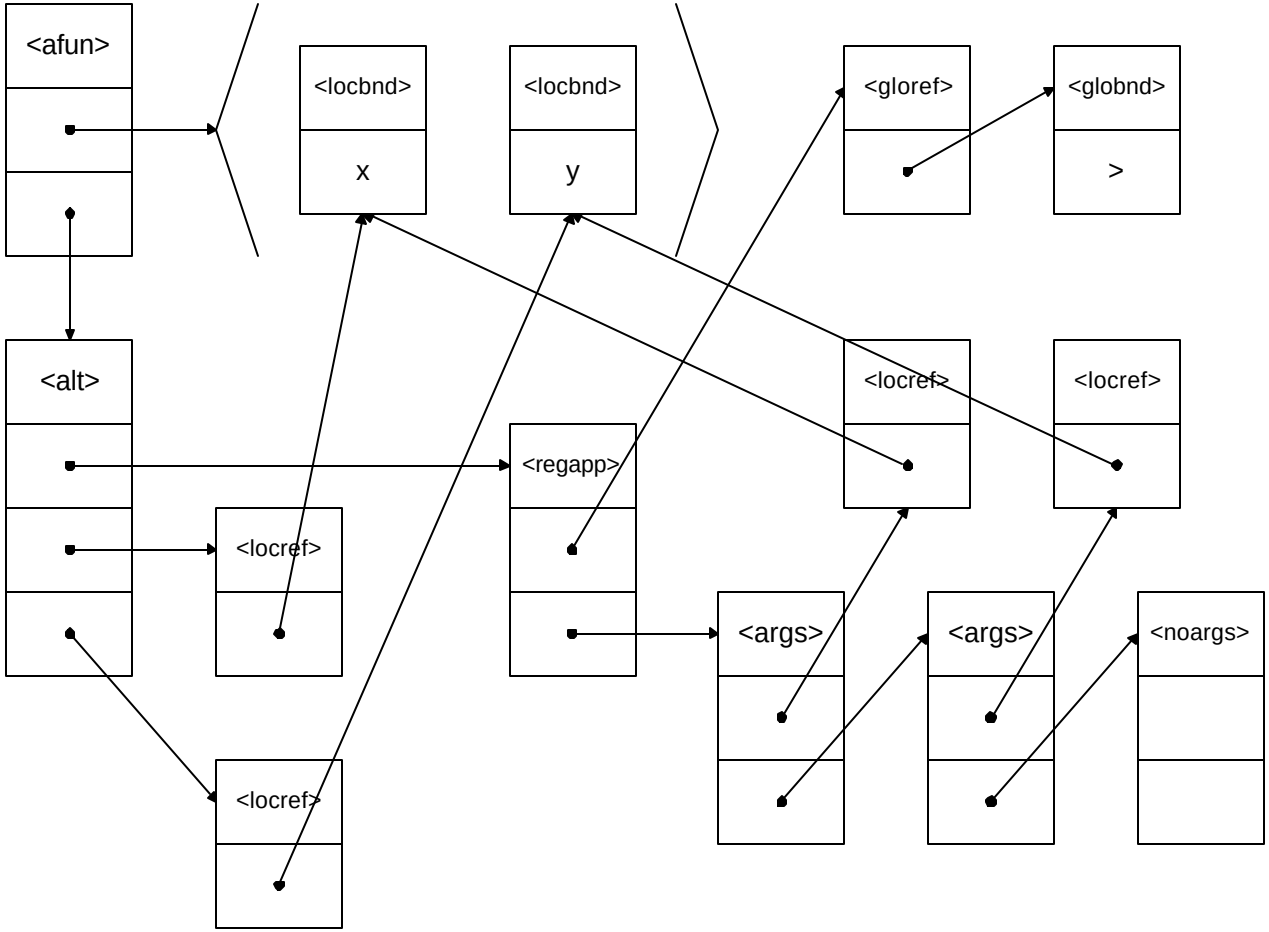
AST Classes

- `<binding>`
- `<function>`
- `<constant>`
- `<reference>`
- `<assignment>`
- `<alternative>`
- `<sequential>`
- `<application>`
- `<fix-let>`
- `<argument-list>`
- `<locals>`
- `<bind-exit>`
- `<unwind-protect>`
- `<monitor>`

- Categorization
  - Local and global
  - Types of functions

Example AST

```
(fun (x y) (if (> x y) x y))
```



Sexpr to AST Conversion

- Analogous to interpretation but produces AST instead of performing evaluation
- `objectify` takes sexpr and static environment as arguments and returns AST
- Magic bindings are defined for each special form and trigger custom converters

AST Interpretation

- `eval` takes an AST object and an environment and interprets the object
- AST conversion has already performed some of the interpretation needed in a naïve interpreter
- Use fast interpreter environments

Goo Runtime

- Objects
- Functions
- Tagging

Goo Objects

- Type-based object system
- Extensible dynamic type system

Goo Functions

- Dispatch cache
  - Tree of association list
    - Traits as keys
    - Subtrees or methods as values
  - Supports singletons with secondary dispatch
  - Folds slot access into cache by storing slot offset as values

Tagging

- Tagging/Boxing scheme hidden with macros
- Uses low order two tag bits
- Encodes four cases:
  - `00` -- pointer
  - `01` -- `<int>`
  - `10` -- `<chr>`
  - `11` -- `<loc>`

Compilation to C

- AST transformations
- Name mangling
- Representations
- Expressions
- Tail calls

Code Walking

- Destructive graph walker

```
(dm update-walk! (g|<fun> o args|...)
  (for ((prop (object-props o))
    (def x ((prop-getter prop) o))
    (when (isa? x <program>)
      ((prop-setter) (apply g o args) o)))
  o)
```

Boxing

- Remove assignments in favor of side-effects
- Box is
  - Created with `make-box` and
  - Accessed with `box-value(-setter)`
- Make analysis simpler (SSA)
- Allows for a flattened closure environment representation

Boxing Walk

```
(dm insert-box! (o|<program>)
  (update-walk! insert-box! o))

(dm insert-box! ((o <local-reference>)
  (if (binding-mutable? (reference-binding o))
      (new <box-read> box-reference o)
      o))

(dm insert-box! (o|<local-assignment>)
  (isa <box-write>
     assignment-reference (assignment-reference o)
     assignment-form      (insert-box! (assignment-form o))))

;; ...
```

Closure Flattening

- C does not support nested functions
- Lambda-lifting migrates lambda's toward the exterior in such a way there are no more interior lambda's
- Basic idea is
  - Transform lambdas into flat-funs which have flat environments
  - Flat environments record all free variables and assign canonical ordering
  - Use `lift!`

Flat Function Example

```
(df f (x) (fun (y) x))

(df fun-1 (m y) (env-elt m 0))
(df f (m x) (fab-fun fun-1 x))
```

- Environments are flattened
  - All closed over variables regardless of nesting are collected
  - Their position in this list defines an offset
- Environment accessed through closure which is passed through in calling convention argument `m`

Collecting Top Level Inits

- Pull out nested functions and quotations so that they are top level initialized
- Create and assign these objects to gensym created anonymously named bindings
- Depart from LiSP by having the scope of top-level initialization be at the top-level-form instead of at the whole file granularity
  - `extract-things!`

Collecting Top Level Initializations Example

```
(df boo (x) (lst (+ x 1) ‘(1)))
(df hoo () 1)
==>
(dv lit-1 (%ib %1))
(dv lit-2 (%pair (%ib %1) %nil))
(df boo (x) (lst (+ x lit-1) lit-2))
(dv lit-3 (%ib %1))
(df hoo () lit-3)
```

Collecting Temporary Variables

- C does not support nested functions and more importantly doesn’t support nested variables with same names
  - `(fun (x) (let ((x (if (== x nul) 0 x))) (+ x 1)))`
- Must remove name conflicts
  - `gather-temporaries!`
  - `(fun (x) (let ((x-1 (if (== x nul) 0 x))) (+ x-1 1)))`

Ready to Output C

- AST graph sufficiently transformed
- Basically a pretty printing exercise
- Need to tie down C runtime hooks

Name Mangling

- Goal reversible for demangling
- Use uppercase characters to encode C

```
- => _
! => X
$ => D
...
```

- Module prefix
- Renamed local variables

C Runtime

- Basic Types
- Primitives
- Calling Convention
- Non Local Exits
- Boxes
- GC
- Symbol Table
- Printing
- Performance Considerations

Basic Types

```
typedef void*           P;
#define PNUL            ((P)0)
typedef float           PFLO;
typedef long            PINT;
typedef unsigned long   PADR;
typedef char            PCHR;
typedef unsigned long   PLOG;
typedef FILE*           PPORT;

typedef union {
  PINT i;
  PFLO f;
} INTFLO;
```

Primitives

- Arithmetic
  - Macros
  - `<flo>`
- Objects
  - Allocation
  - Cloning
  - Slot access
- Basic types
  - `<vec> <lst> <str>`
- Functions
  - Closures
    - `FUNINIT`
    - `FUNSHELL`
    - `FUNFAB`
- I/O

Calling Convention

- Unknown calls
  - `<gen>` and `<met>` and otherwise cases
  - Congruency checking
    - `CHECK_TYPE`
    - `CHECK_ARITY`
- Temporary argument stack
- Cons up optional arguments
- `%apply`
  - Also `%mep-apply`

CALLN

```
P CALLN (P fun, int n, ...) {
  int i, j;
  P traits = YPobject_traits(fun);
  if (traits == YLmetG_traits) {
    int arity = FUNARITY(fun);
    P   specs = FUNSPECS(fun);
    int naryp = FUNNARYP(fun);
    va_list ap; va_start(ap, n);
    for (i = 0; i < arity; i++) {
      P arg = va_arg(ap, P); PUSH(arg);
      CHECK_TYPE(arg, Phead(specs));
      specs = Ptail(specs); }
    if (naryp) {
      int nopts = n - arity;
      P   opts  = Ynil;
      for (i = 0; i < nopts; i++)
        a[i] = va_arg(ap, P);
      for (i = nopts - 1; i >= 0; i--)
        opts = YPpair(a[i], opts);
      PUSH(opts); }
    CHECK_ARITY(naryp,n,arity);
    va_end(ap);
    return (FUNCODE(fun))(fun);
  } else if (traits == YLgenG_traits) {
    /* ... */
  } else {
    return CALL1(Yunknown_function_error, fun); } }
```

Non Local Exits

- Uses C's `longjmp`
- C Structures
  - `bind_exit_frame`
  - `unwind_protect_frame`
- C Support routines
  - `nlx_step`
  - `do_exit`
- Conversion using thunks
  - `with_exit`
  - `with_cleanup`

Example Non Local Exit

```
(lab (ret) (ret 1))

=>

(%with-exit (fun () (ret 1)))

;; using

P with_exit (P fun) {
  BIND_EXIT_FRAME frame = MAKE_BIND_EXIT_FRAME();
  P exit = YPmet(&do_exit, YPpair(YLanyG, Ynil), YPfalse, YPib((P)1),
                           FABENV(1, frame));
  if (!setjmp(frame->destination))
    return CALL1(fun, exit);
  else
    return frame->value;
}
```

Boxes

- C support
  - BOXVAL
  - BOXFAB
- Can remove boxes if in same environment

GC

- Boehm collector
  - Written in C
  - Public domain
  - Conservative
- Only need GC_alloc

Symbol Table

- Register during C definition
- Build up for
  - Mapping over native bindings
    - Used for integrating with interpreter environment
  - Debugging using original names
  - Reverse mapping from address to name

Printing

- Builtin printing knows object format
  - `des`
  - `print`
- Callable from `GDB`
- Indispensable for low level debugging

Performance

- Inlining
  - `INLINE` macro
  - Primitives
- Specialize a few `CALL`*n* versions
- Stack allocation
  - Optional arguments

Basic C File Breakdown

```
(dm generate-c-program (out e prg ct-env)
  (generate-header out e)
  (generate-global-environment out ct-env)
  (generate-quotation-forwards out (program-quotations prg))
  (generate-function-forwards out (program-definitions prg))
  (generate-function-bodies out (program-definitions prg))
  (generate-main out (program-form prg))
  (generate-trailer out)
  prg)
```

Actual C File

```
/* PROTO 2 C $REVISION: 0.1 $ */
#include "proto.h"
/* GLOBAL ENVIRONMENT: */
DEF(YOfun_specs,"@fun-specs");
/* ... */
/* FORWARD QUOTATIONS: */
DEFLIT(lit_739);
/* ... */
/* FUNCTIONS: */
extern P YPPtraits (P);
/* ... */
LOCFOR(fun_loop_91);
/* ... */
FUNFOR(Ytraits_ordered_parents);
/* ... */
```

```
/* FUNCTION CODES: */
P YPPtraits(P owner_) { /* ... */ }
/* ... */
P Y___main___() { /* ... */ }
/* ... */
int main(int argc, char* argv[]) {
  YPinit_world(argc, argv);
  (((P)Y___main___()));
  return(0);
}
```

Expressions

- Utilize C's expression oriented constructs
  - `T ? C : A`
  - `(E1, E2, …, En)`
- Avoids creating intermediate temporaries and/or register allocation
- Unfortunately makes debugging difficult

Example of Expressions

```
(seq (doit) (if (done?) #t #f))

==>

(CALL0(Ydoit), CALL0(YdoneQ) !== Yfalse ? Ytrue : Yfalse))
```

Primitives

- Used for bootstrapping and efficiency
- Called with normal `C` calling convention
  - No Proto argument stack
  - Arguments are always coerced to `P`
- Code only
- Examples:
  - C runtime primitives like `%i+`
  - Booting primitives like `@len`

Example Primitive

```
(dl len (x)
  (if (%empty? x) (%raw 0) (%i+ (%%len (%tail x)) (%raw 1))))

==>

P YPPlen(P x_) {
  return (((P)YPemptyQ((P)x_) != YPfalse)
          ? (P)0
          : (P)YPiA((P)YPPlen((P)YPtail((P)x_)), (P)1));
}
```

Functions

- Arguments passed on special stack
  - Suboptimal but very easy for C runtime
  - Called through using congruence checker trampoline
- Function pointer passed in only required argument
  - Used for accessing closed over variables
- Temporaries declared as C local variables

Example Function

```
(df not ((x <any>) => <log>) (%bb (%eq? x #f)))

==>

FUNCODEDEF(Ynot) {
  ARG(x_);

  return (P)YPbb((P)(P)YPeqQ((P)x_,(P)YPfalse));
}
```

Tail Calls and Loops

- Naïve C emission misses causes inefficient stack growth
- Simple approach can regain some
  - Detect self-recursive calls
  - Declare
    - Label at beginning of body
    - Argument shadow variables
  - Turn self-recursive calls into
    - Argument xfers +
    - Goto

Example Loop

```
(df @fill ((x <lst>) (f <fun>) => <lst>)
  (rep loop ((x x))
    (if (@empty? p) x (seq (set (@head p) f) (loop (@tail p))))))

==>

FUNCODEDEF(fun_loop_232) {
  ARG(x_);
  P res, a1;
loop:
  res = (((P)YOemptyQ((P)CHKREF(Yp)) != YPfalse)
         ? x_
         : ((P)YOhead_setter((P)FREEREF(0),(P)CHKREF(Yp)),
            (a1=(P)YOtail((P)CHKREF(Yp)),x_=a1,PNUL)));
  if (res == PNUL) goto loop; else return res;
}
```

Advanced Topics

- Alternative Bootstrapping
  - Compiler based
  - Minimal boot that reads virgin code through
- Compilation to C
  - Separate compilation
  - Static creation of data
- C runtime
  - Tail calls
  - Global register assignment with gcc

Goo Status

- Ready for first public release
- Simple c-based dynamic compilation
- Almost no compiler optimizations

Future

- Improved runtime
  - Faster calling convention
  - Fast subtype tests
- Compiler optimizations
  - Dynamically created dispatchers
  - Inlining
  - Constant folding

Reading List

- Queinnec: LiSP