shux

A physics simulation language for Lagrangian physics
Our Team

- Lucas Schuermann: Manager, physics dude
- John Hui: Language guru
- Mert Ussakli: Code slave
- Andy Xu: System architect
Inspiration

Growing field of particle-based numerical physics solvers
- Fluid dynamics
- Granular flow
- Deformables
Main features

● Everything good about C, redesigned for easy implementation of particle-based Lagrangian physics solvers

● Mostly immutable (unless you cheat) types for concurrency

● Simple functional syntax
  ● Maps, filters, lambdas
  ● Namespaces
  ● Lookback and generators (what are those?!)

● Easy bindings to OpenGL through extern declarations
Immutability

int x = 1;

var int y = 2;

y = 3; /* this is OK */

x = 5; /* this is not OK */

Allows for guaranteed safety in concurrent settings!
Map!

```plaintext
kn addOne(int x) int {
    x+1
}

kn main() int {
    int[5] x = [1,2,3,4,5];
    int[5] xPlusOne = x @ addOne;
    0
}
```
Filters

```c
kn lessThanThree(int x) bool {
    x < 3
}

kn main() int {
    int[5] array = [2,3,4,5,6];
    int[] filtered = array :: lessThanThree 0
}
```
\text{λs}

(low key type inferred)

kn main(int {
    int[5] x = [1, 2, 3, 4, 5];
    bool[5] b = x @ (int i) -> {
        i % 2 == 0
    }
})
ns constants = {
    ns physical_params = {
        let vector <2> grav = (0.0, -9.81);
    }
    ns = solver_params = {
        let scalar dt = 0.001;
    }
}

scalar y = constants \rightarrow physical_params \rightarrow grav[1];
Everything is an expression

int y = 2
int x = if y == 2
The lookback feature and generators

gn fib() int {
    int y = (y..1 : 1) + (y..2 : 1);
    y
}

kn main() int {
    int fib5 = do 5 fib( );
    0
}
Native LLVM OpenGL Binding

extern graphics_init();
extern graphics_loop(scalar[] points_buf);

kn main() int {
    graphics_init();
    ...
    graphics_loop(...);
    0
}


define void @graphics_do_update() {
entry:
call void @clear(i32 16640)
call void @LoadIdentity()
call void @ortho(float 0.000000e+00, float 8.000000e+02, float 0.000000e+00, float 6.000000e+02)
call void @color4f(float 0x3FC99999A0000000, float 0x3FE3333340000000, float 1.000000e+00, float 1.000000e+00)
call void @begin(i32 0)
call void @vertex2f(float 0.000000e+00, float 0.000000e+00)
call void @end(i32 0)
call void @glutSwapBuffers()
ret void
}

define void @graphics_init() {
entry:
%argc = alloca i32
store i32 0, i32* %argc
call void @glutInitWindowSize(i32 800, i32 600)
LLVM<>OpenGL
Simple Demonstration
Workflow

- To get all of this working in LLVM, we implemented a pipeline with several layers of translation.
- Goal is to convert code with semantics most distant from C as close as possible to C before generating LLVM IR.
How Crazy Were We?

See next slide...
and lit =
  | LitInt of int
  | LitFloat of float
  | LitBool of bool
  | LitStr of string
  | LitK of lambda
  | LitVector of expr list
  | LitArray of expr list (* include optional type annotation here? *)
  | LitStruct of string list * struct_field list (* should this be more sophisticated? *)

and struct_field = StructField of string * expr

and expr =
  | Lit of lit
  | Id of string list
  | Lookback of string list * int
  | Binop of expr * bin_op * expr
  | Assign of expr * expr
  | Call of string list option * expr list
  | Uniop of un_op * expr
  | LookbackDefault of expr * expr
  | Cond of expr * expr * expr (* technically Ternop *)
  | Access of expr * string

and stmt =
  | VDecl of bind * expr option
  | Expr of expr

type fn_decl = {
  fname : string;
  fn_typ : fn_typ;
  ret_typ : typ option;
  formals : bind list;
  body : stmt list;
  ret_expr : expr option;
}
type slit =
| SLitInt of int
| SLitFloat of float
| SLitBool of bool
| SLitStr of string
| SLitKn of slambda
| SLitArray of sexpr list
| SLitStruct of string * ((string * sexpr) list)

and sexpr =
| SLit of styp * slit
| SID of styp * string * sscope
| SLookback of styp * string * int
| SAccess of styp * sexpr * string
| SBinop of styp * sexpr * sbin_op * sexpr
| SAssign of styp * sexpr * sexpr
| SKnCall of styp * string * (sexpr * styp) list
| SGnCall of styp * string * (sexpr * styp) list
| SExCall of styp * string * (sexpr * styp) list
| SLookbackDefault of styp * int * sexpr * sexpr
| SUnop of styp * sun_op * sexpr
| SCond of styp * sexpr * sexpr * sexpr
| SLoopCtr (* CLoopCtr, useful for recursion *)
| SPEek2Anon of styp
| SExprDud

and slambda = {
  slret_typ : styp;
  slformals : sbind list;
  slinherit : sbind list;
  slllocals : sbind list; (* no lookback, const-ness not enforced *)
  slbody : (sexpr * styp) list;
  slret_expr : (sexpr * styp) option;
}
type clit =
  | CLitInt of int
  | CLitFloat of float
  | CLitBool of bool
  | CLitStr of string
  | CLitArray of clit list
  | CLitStruct of (string * clit) list
  | CLitDud

type cexpr =
  | CLit of ctyp * clit
  | CID of ctyp * string
  | CLoopCtr (* access the counter inside a CLoop *)
  | CPeedAnon of ctyp (* access the temp value of a CBlock *)
  | CPeed2Anon of ctyp (* access the temp value of a CBlock *)
  | CPeed3Anon of ctyp (* access the temp value of a CBlock *)
  | CBinop of ctyp * cexpr * cbin_op * cexpr
  | CAccess of ctyp * cexpr * string
  | CAssign of ctyp * cexpr * cexpr
  | CCall of ctyp * string * cstmt list
  | CExprCall of ctyp * string * cstmt list
  | CUnop of ctyp * cun_op * cexpr
  | CExprDud

and cstmt =
  | CExpr of ctyp * cexpr
  | CCond of ctyp (* if *) cstmt (* then *) cstmt (* else *) cstmt
  | CPushAnon of ctyp * cstmt
```plaintext
type lltyp =
  | LLBool (* i1 *)
  | LLInt (* i32 *)
  | LLDouble (* double_type *)
  | LLConstString (* name and content, only used for representing strings, simply i8* *)
  | LLArray of lltyp * int option (* inside formal we need to declare int*; inside local we declare int[len] *)
  | LLStruct of string
  | LLVoid (* only used for declaring function types *)

type llllit =
  | LLLitBool of bool
  | LLLitInt of int
  | LLLitDouble of float
  | LLLitString of string
  | LLLitArray of llllit list
  | LLLitStruct of llllit list

type llreg =
  | LLRegLabel of lltyp * string (* register can store a name and an lltyp value *)
  | LLRegLit of lltyp * llllit
  | LLRegDud
```
Case Study: Lookback

gn bar(int a, int b) int {
    int x = a + x .. 1
    : 3;
    int y = b + y .. 2
    : 2;
    x+y
}

struct gn_bar = {
    int ctr; int[2] a; int[2] b;
}

kn bar(struct gn_bar gns) int {
    gns.x[gns.ctr] = gns.a[gns.ctr] +
    (ctr <= 1) ? gns.x[(gns.ctr-1)%2] : 3;
    gns.y[gns.ctr] = gns.b[gns.ctr] +
    (ctr <= 2) ? gns.y[(gns.ctr-2)%2] : 2;
    gns.x[gns.ctr] + gns.y[gns.ctr]
}
Testing Environment
A large suite of automated unit tests were used to thoroughly test every semantic aspect of the language.

When changes were made to frontend or code was added for lower level translations, the suite was run.

Over 150(!) tests allowed us to rigorously verify syntax and steps through CAST generation.
The Good News: What Works

We have a fully implemented:

- Frontend
- Semantic checker
- AST to SAST translation
- SAST to CAST translation
- CAST to LLAST… (to be continued)
- Translation from LLAST to LLVM
Frontend

- Fully tested and robust parser
- Handles a number of edge cases discovered through tests
- Completed very early on in development to ensure testing further down the line
Semantic Checker

- shux has a strict type checking system, but at the same time maps, filters and generators, expressions complicate type-checking
- Lambdas have type inference
- The goal for the strict type system was readability and ease of translation
- semant.ml is 719 lines of OCaml
- Lookback values are an exception to the rule
  - `int x = x..1; /* accessed while being defined */`
AST to SAST

- Makes all types in all expressions explicit. Important for translating an expression-based language.
- Does heavy-lifting for further stages of translation
  - Lookback values
  - Hoisting declarations above expressions in functions and lambdas
- Get rid of option types.
- Separating semantics:
  - kernel calls vs generator calls
  - float operands and int operands
SAST to CAST

@John the orator
CAST to LLAST

@John the orator
LLAST To LLVM

- All the LLVM binding specific usage is abstracted in previous levels of translation
- Hide the registers from levels above
- Only operate on stack variables
- Passing by reference in LLVM
  - Array
  - String
  - Structs
- Using global namespace
The Bad: Or, The Perils of Ambition: Real-World Tests

- Multi-stage pipeline led to many, many, many blocking portions of development or propagating work from changes
- Time was spent fleshing out an amazing sheer volume of code
- We fell a bit sort on demos to show because we were more focused on designing, implementing, and testing a full pipeline
Future Work

- Testing and bugfixes for last two stages of the pipeline, relating to:
- More fully compiled complex usages of the language to generate results
- Finishing filters
- More robust standard library: further graphics calls, gridding, vector operations baked in (dot, matmul, etc)