Slang: A discrete event simulation language

The Team
Olivia Byer
Mauricio Castaneda
Josh Itwaru
Dina Lamdany
Tony Ling

Motivation
The goal was to create a language that would allow the user to run simulations of events. The language exists for simulation in languages such as Verilog, and we wanted to expand the concept to apply to situations such as queueing problems.

Overview
Slang is a programming language designed to simulate events through the use of data structures. The language is compiled and interpreted, and its syntax is designed to be concise and efficient.

Compiler Architecture
The compiler's implementation is in C++.

Lessons Learned
Some lessons learned in the process include:

- The importance of user feedback in refining the language.
- The challenges of integrating simulation with real-world data.
- The potential for Slang to be used in a variety of fields.

Tutorial: Features of a Slang program

How to Compile and Run a Slang Program

1. Compile the program.
2. Run the program.
3. View the output.

Sample Program 1

Sample Program 2

Sample Program 3
Slang: A discrete event simulation language
The Team
Olivia Byer
Mauricio Castaneda
Josh Itwaru
Dina Lamdany
Tony Ling
Motivation

Our goal was to create a language that would allow the programmer to simulate one-time and recurring events. This framework exists for hardware in languages such as Verilog, and we wanted to expand this model to apply to situations such as queuing problems.
Overview

In Slang, the programmer can schedule events at discrete times in an event queue through the use of delay statements. Additionally, the placement of statements in init or always blocks allows for both one-time and recurring events.

Slang utilizes static scoping and is strongly typed.
Tutorial: Features of a Slang program

• Program Structure
  • All programs must have a main
  • User-defined functions can be declared above the main
  • init and always blocks exist inside the main - init blocks run once upon execution, while always blocks run on a continuous loop

• Data Types and Variables
  • Available data types are string, int, float, boolean, and array. Void can also be returned by functions.
  • Variables defined in main are global to init and always blocks but must be passed into functions as parameters

• Programmatic Features
  • for and while loops and if statements are all available features, in c-like syntax
  • Delay statements make program time move forward by a specified integer amount
  • Unary and binary operators are available, see manual for specifications
functions declared above the main

```cpp
func void helloworld()
{
    print("hello world");
}

main()
{
    init()
    {
        print("Welcome to the demo!");
        #2
        Terminate;
    }
    always()
    {
        #1
        helloworld();
    }
}

one main function per program

at least one init block that runs once upon start up of the program

Delay statements that add time to the program clock

An optional terminate statement to end program execution
always blocks which run on a loop until program terminates
How to Compile and Run a Slang Program

1) make clean
2) make
3) ./compiler < [path to your .sl file]
4) g++ output.cpp
5) ./a.out
Sample Program 1

func int fib(int n){
    /* Base Case */
    if(n==0){return 0;}
    if(n==1){return 1;}

    int prevPrev=0;
    int prev=1;
    int result=0;
    int i=2;

    /* Calculate Results */
    for(i=2; i<=n; i++){
        result=prev+prevPrev;
        prevPrev=prev;
        prev=result;
    }

    return result;
}

main(){
    init{
        #1
        int fib=fib(7);
        print(fib);
    }
}
main(){
    int prevPrev=0;
    int prev=1;
    int result=0;

    init{
        #7 print(result);
        Terminate;
    }

    /* Loop to calculate numbers*/
    always{
        #1
        result=prev+prevPrev;
        prevPrev=prev;
        prev=result;
    }
}
Compiler Architecture

- Scanner
  - Slang Source File
  - Tokens

- Parser
  - Abstract Syntax Tree

- Semantic
  - Semantically Checked Abstract Syntax Tree

- Pretty C
  - Intermediate Representation
  - C++ Code

- Generator
  - C++ Code
Compiler Architecture

helloworld.sl

Scanner and Parser

([], [], [Init [Event (0, [Expr (StringLit "hello world")])]])

Semantic Check

Prog ([], ([], [SInit [SEvent (0, [SSExr (SStringLit ("hello world", Datatype String))])]]))
Compiler Architecture

Semantically Checked Program

Pretty C

Pretty_c.Pretty_c ([], []),
[Pretty_c.Time_block (Pretty_c.Link "init_0", []),
[Pretty_c.Time_struct (Pretty_c.Time_struct_name "init_0_block_0", 0,
  Pretty_c.Link "init_0",
  [SSExpr (SStringLit ("hello world", Datatype String)))]),
Pretty_c.Main
([Pretty_c.Time_struct_obj (Pretty_c.Time_struct_name "init_0_block_0",
  Pretty_c.Link "init_0")],
[Pretty_c.Link "init_0"], [])
Compiler Architecture

Intermediate Representation from Pretty C

Code Generation

138 Lines of C++ Code generated from just 16 lines of Slang Code
Lessons Learned

1) Communicate and delegate effectively so that no one is doing duplicate work and so that people don't feel that they are taking on all the work.

2) Make sure you fully understand the basics. Do this by writing code rather than just reading code.

3) Test your code in small sections rather than being stuck with a really confusing OCaml compiler error message that could apply to any one of many lines of code.

4) Start earlier than you think you should. Bugs in later sections can lead to changes having to be made in earlier sections, so you aren't necessarily done with a part even when you think you are.
The Team
Olivia Byer
Mauricio Castaneda
Josh Itwaru
Dina Lamdany
Tony Ling

Motivation
Our goal was to create a language that would allow the compiler to understand time and resource constraints. Languages like ML and Standard ML, while highly expressive, do not handle these constraints well. A new language was needed to support these constraints and be robust enough to apply to real-world situations such as operating systems.

Overview
Slang is a new language that addresses the need for a language that can understand and account for time and resource constraints. It is designed to be a high-level, high-performance language that can be used to develop real-world systems.

Lessons Learned
1. Understanding of time constraints is crucial for real-world applications.
2. Resource management is essential for efficient system development.
3. A language that can handle these constraints is necessary for the development of complex systems.

Compiler Architecture
A compiler architecture is developed to support the language. It includes modules for parsing, code generation, and optimization.

Sample Program 1
The sample program demonstrates the use of the language in a simple task, such as file processing.

Sample Program 2
The sample program shows the language's ability to handle more complex tasks, such as resource allocation.

Tutorial: Features of a Slang Program
Slang programs can be written in a high-level, expressive manner. They can be compiled and run in a real-time environment, demonstrating the language's potential for real-world applications.