GAIL: A DESIGN AND IMPLEMENTATION OF A CONSTRAINED GUARDED ACTION INTERMEDIATE LANGUAGE SUITABLE FOR REWRITE-BASED OPTIMIZATION

Tim Zwiebel
Northwestern University

Overview

GAIL

Rewriting

Code Generator

User Interface

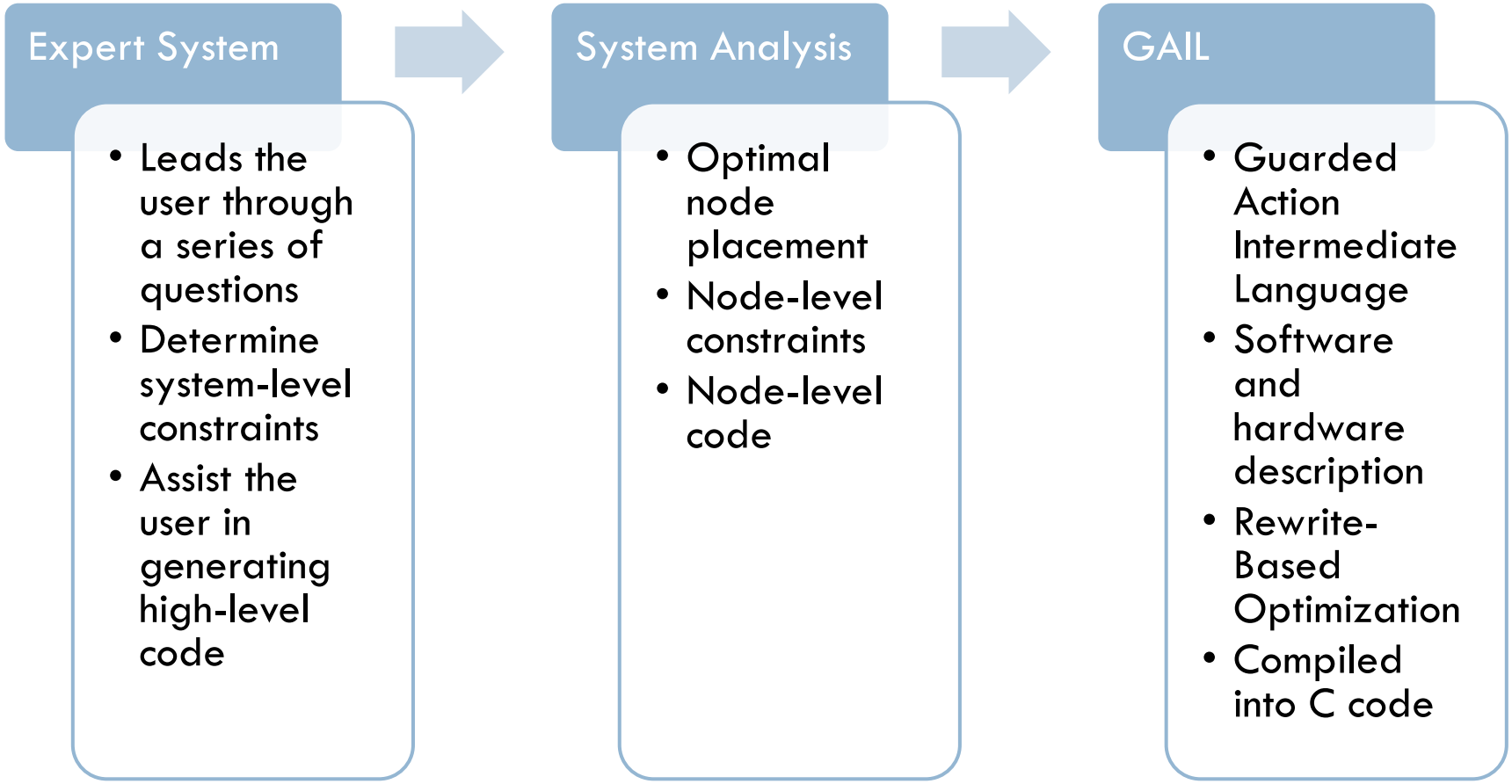
Future Work

Conclusion

Wireless Sensor Networks

- Archetype-Based Synthesis
- Wireless Communication
- Battery Powered
- Complex Wireless communication protocols

ABSYNTH Project



Overview

GAIL

Rewriting

Code Generator

User Interface

Future Work

Conclusion

GAIL

- Guarded Action Intermediate Language
- Designed for use in Wireless Sensor Networks (runs on constrained hardware platforms)
- Language is constrained to facilitate program analysis
- Programs contain both hardware and software
- Designed to allow rewrite-based optimizations on both hardware and software

GAIL Programs

- Programs are ((Def …) (GA …))
  - Def=Variable & Hardware Definitions
  - GA = Guarded Actions
- All variables and hardware is defined and initialized in the hardware definition section
- Types:
  - Boolean, Boolean Queues
  - Scalar, Scalar Queues
  - Analog and Digital Inputs
  - Analog and Digital Outputs

Queues

- Queues are statically allocated
- Queues can become full, so each queue has a policy to handle this case
- Policies
  - DROP: items added to a full queue are discarded and the queue remains the same
  - DISPLACE: items added to a full queue displace an item already in the queue (e.g. if an item is added to the front, an item is popped off the back)

Guarded Actions

- Guards are boolean expressions
- When guards evaluate to TRUE, the actions are triggered
- Guarded actions have constraints to determine when to evaluate the guard
- Example Constraint: (time-constraint 0 3000)
- Other Constraints: variable-constraint, guard-constraint, messages, interrupts

Evaluation of Guarded Actions

- Guards are evaluated according to their constraints
- When a guard is evaluated to TRUE, the entire set of actions is evaluated
- Hardware is sampled once for the entire set of actions
- Functions with side effects must be at the top level of an action
- When evaluating a set of actions, all expressions that do not have side effects are evaluated before those that do
- Expressions that have side effects are evaluated in order

Overview

GAIL

Rewriting

Code Generator

User Interface

Future Work

Conclusion

Rewriting

- Optimizations are performed via rewrite rules
- Uses PLT Redex, part of PLT Scheme
- Rewrite rules are called reduction cases and a set of rewrite rules is a reduction relation

PLT Redex Example – Language Definition

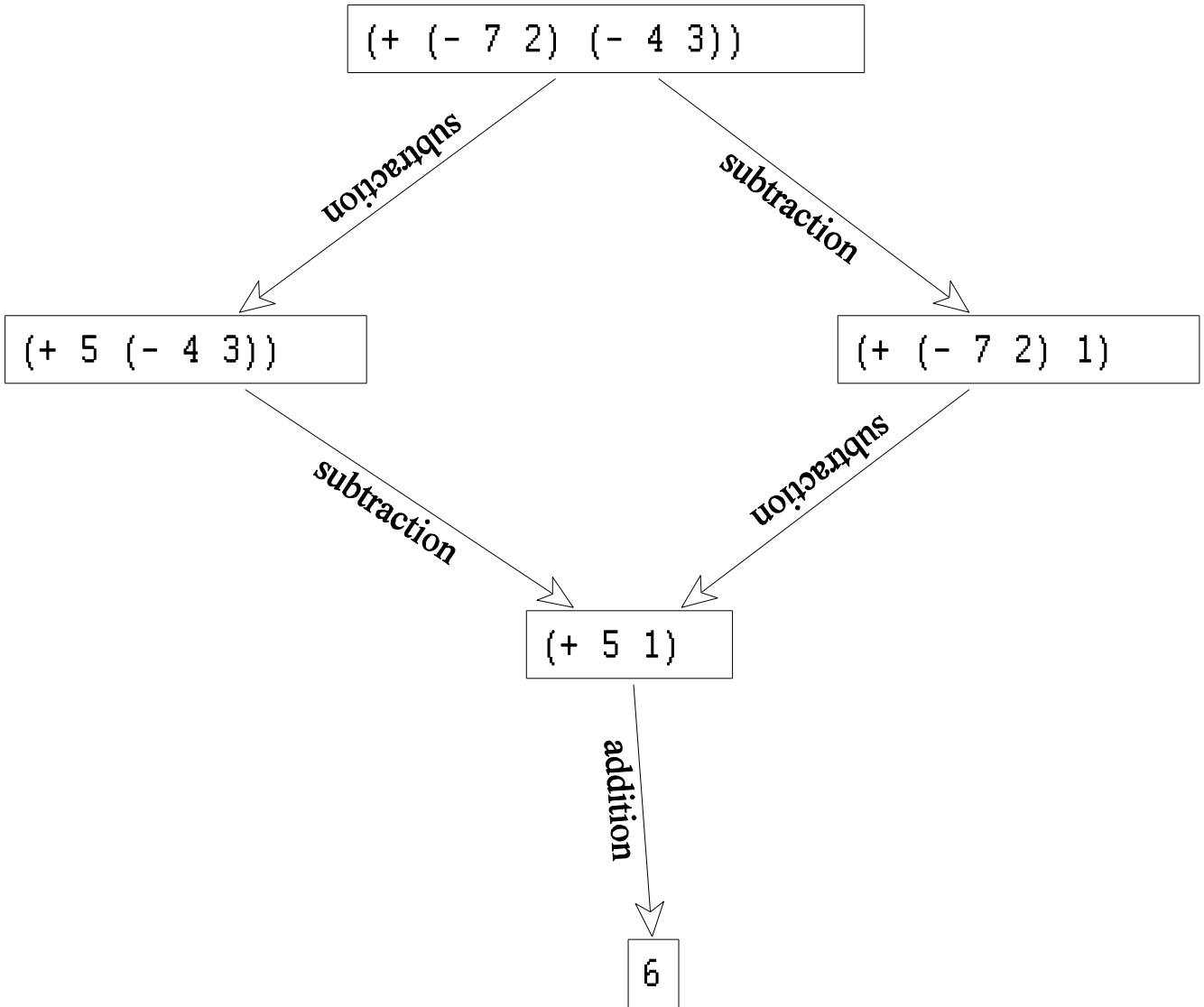

```
(define-language simple-lang
  (exp number
       (+ exp exp)
       (- exp exp))
  (C hole
     (any ... C any ...)))
```

PLT Redex Example – Rewrite Rules


```
(define rewrite
  (reduction-relation
   simple-lang
   (--> (in-hole C (+ number_1 number_2))
        (in-hole C ,(+ (term number_1) (term number_2)))
        "addition")
   (--> (in-hole C (- number_1 number_2))
        (in-hole C ,(- (term number_1) (term number_2)))
        "subtraction")))
```

PLT Redex Example - Traces

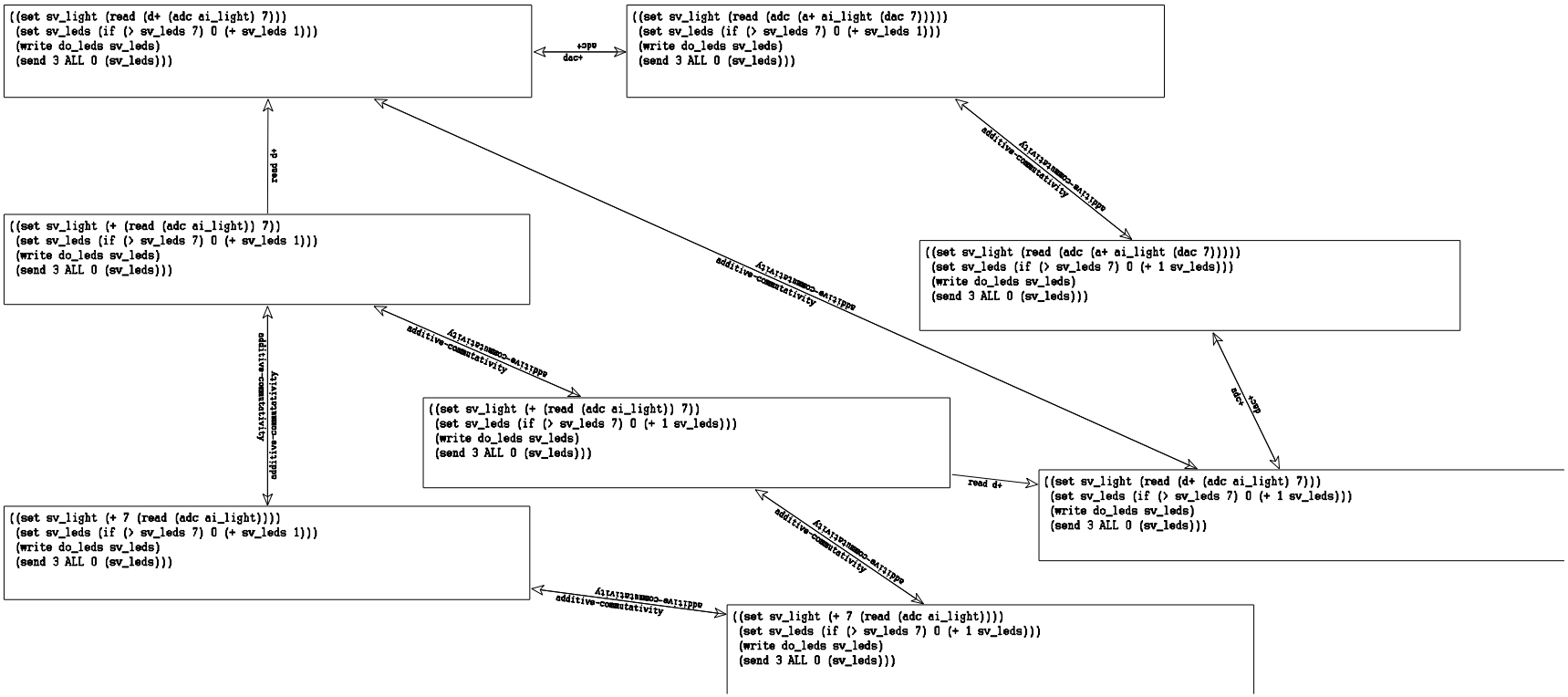

```
(traces rewrite (term (+ (- 7 2) (- 4 3))))
```



Joint Hardware-Software Optimization


```
(--> (in-hole C (+ (read di-exp_1) number_1))
     (in-hole C (read (d+ di-exp_1 number_1))
     "read d+")
(--> (in-hole C (+ (read ai_exp_1) number_1))
     (in-hole C (read (a+ ai-exp_1 (dac number_1))))
     "read a+")
```

Example Rewriting



This graph is unreadable since it won't fit on one slide

Objective Functions

- The rewrite system generates equivalent programs
- An objective function is used to determine the "best" program
- GAIL is constrained, so program analysis is easier
  - Maximum stack depth
  - Memory usage
  - Program code size
  - Estimates of power/energy use

Overview

GAIL

Rewriting

Code Generator

User Interface

Future Work

Conclusion

Code Generator: LISA

- LISA is a Java-based compiler generator
- It uses attribute-based grammars
- Generates a scanner, parser, and evaluator
- Attributes are Java types
- Java assignment statements in the formal grammar determine the values for the attributes
- LISA can automatically determine inherited vs. synthesized attributes

C Code

- Initialize variables
- Main Loop
  - Wait on a semaphore
- Constraints place a function pointer in a task queue, then post on the semaphore

LISA Example - Scanner


```
language SimpleLang {
    lexicon {
        NUMBER \-?[0-9]+(.[0-9]+)?
        PLUS \+
        MINUS \-
        LP \(
        RP \)
        //space, tab, line feed, carriage return
        WHITESPACE [\ \0x09\0x0A\0x0D]
        ignore #WHITESPACE
    }

...
```

LISA Example - Attributes


```
...

    attributes String PROG.code, EXP.val, EXP.code;

    rule Program {
        PROG ::= EXP compute {
            PROG.code = "#include <stdio.h>\n\nint main() {\n" +
                EXP.code + "\nprintf(\"%f\\n\", " + EXP.val +
                ");\n\nreturn 0;\n}";
        };
    }

...
```

LISA Example - Grammar


```
...

    rule Expression {
        EXP ::= #NUMBER compute {
            EXP.val = getTemp();
            EXP.code = "float " + EXP.val + " = " +
                #NUMBER.value() + ";\n";
        };

        EXP ::= ( #PLUS EXP EXP ) compute {
            EXP[0].val = getTemp();
            EXP[0].code = EXP[1].code + EXP[2].code + "float " +
                EXP[0].val + " = " + EXP[1].val + " + " +
                EXP[2].val + ";\n";
        };

        EXP ::= ( #MINUS EXP EXP ) compute {
            EXP[0].val = getTemp();
            EXP[0].code = EXP[1].code + EXP[2].code + "float " +
                EXP[0].val + " = " + EXP[1].val + " - " +
                EXP[2].val + ";\n";
        };
    }

...
```

LISA Example - Methods

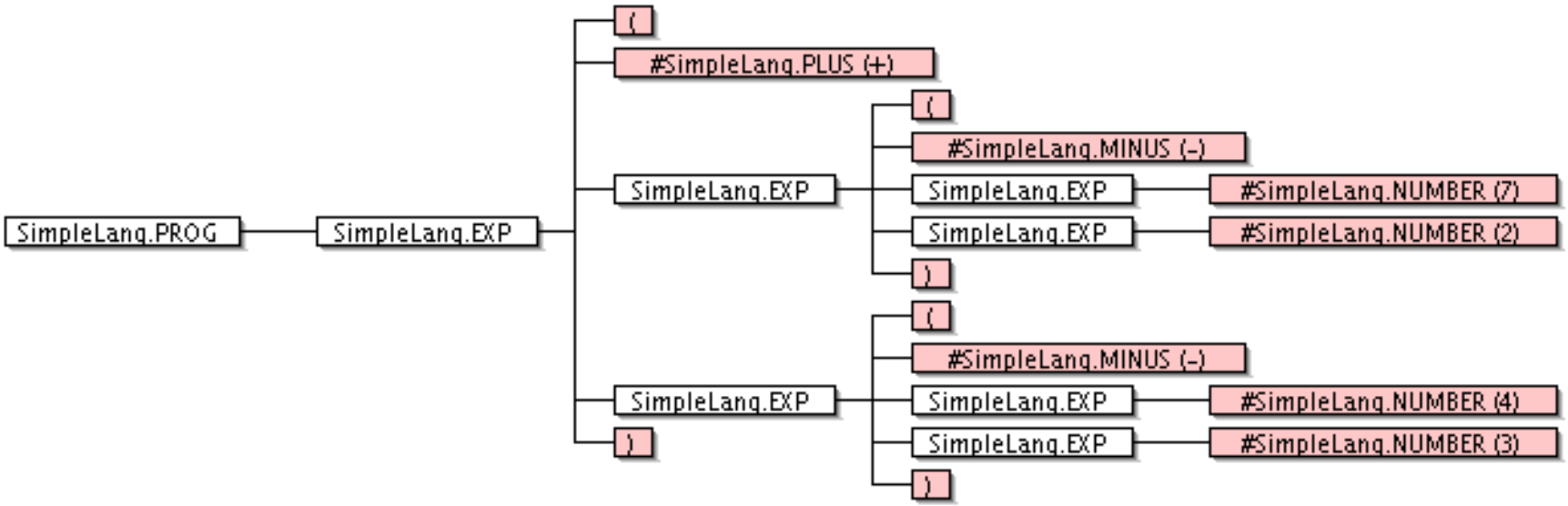

```
...

    method Conversions {
        double stringToDouble(String s) {
            return Double.parseDouble(s);
        }
    }

    method Temps {
        static int tempCount = 1;

        String getTemp() {
            return "temp" + tempCount++;
        }
    }
}
```

LISA Tree


SimpleLang.PROG
SimpleLang.EXP
(
#SimpleLang.PLUS (+)
SimpleLang.EXP
(
#SimpleLang.MINUS (-)
SimpleLang.EXP
#SimpleLang.NUMBER (7)
SimpleLang.EXP
#SimpleLang.NUMBER (2)
)
SimpleLang.EXP
(
#SimpleLang.MINUS (-)
SimpleLang.EXP
#SimpleLang.NUMBER (4)
SimpleLang.EXP
#SimpleLang.NUMBER (3)
)
)

LISA Example – C Code


```
#include <stdio.h>

int main() {
float temp3 = 7;
float temp4 = 2;
float temp2 = temp3 - temp4;
float temp6 = 4;
float temp7 = 3;
float temp5 = temp6 - temp7;
float temp1 = temp2 + temp5;

printf("%f\n", temp1);

return 0;
}
```

Overview

GAIL

Rewriting

Code Generator

User Interface

Future Work

Conclusion

User Interface

- Java Applet
- Webcam
- Demo

Future UI Work

- Graphics to see LEDs, etc.
- Buttons to actuate sensors
- Improved editor
  - Syntax highlighting, line numbers, code completion

Overview
GAIL
Rewriting
Code Generator
User Interface
Future Work
Conclusion

Future Work

- Implementation of data types
  - Scalars
  - Queues
- Communication
  - Multi-hop
- Optimizations
  - More of them
  - Directed search

Future Work

- Verification
  - Accuracy maintained?
  - Power/energy consumption
- Runtime Errors

Overview

GAIL

Rewriting

Code Generator

User Interface

Future Work

Conclusion

Conclusion

- Guarded Action Intermediate Language
- Runs in a constrained hardware environment
- Rewrite system allows joint hardware-software optimization
- Constrained nature of language allows easier program analysis

QUESTIONS?