AbleC: An Extensible C Framework

Ted Kaminski and Eric Van Wyk

University of Minnesota

Pittsburgh, WG2.11, March, 2014

A refresher and introduction.

Extensible languages frameworks

Allow programmers select the features to be used in their programming languages.

- new syntax (notations)
- new semantic analyses
  (error-checking, optimization opportunities)
- new optimizations

Why would anyone want to do that?

Programming language features

General purpose features in, *e.g.*, C or Java

- assignment statements, loops, if-then-else statements
- functions (perhaps higher-order) and procedures
- I/O facilities
- modules
- data: integer, strings, arrays, structs

Domain-specific features

- matrix operations (MATLAB)
- regular expression matching (Perl, Python)
- statistics functions (R)
- computational geometry operations (LN)
- parallel computing (SISAL, NESL, SAC, PQL etc.)

Many similarities, needless differences.

Working with multiple (domain-specific) languages is a headache.

Extensible languages

Allow programmers select the features to be used in their programming languages.

- new syntax (notations)
- new semantic analyses
  (error-checking, optimization opportunities)
- new optimizations

Pick a general purpose host language (*e.g.* ANSI C), extend with domain-specific features.

`myProgram.xc` $\Longrightarrow$ `myProgram.c`

Reliably composable language extensions

- Extension developers work independently.
- Extension users use multiple extensions.
  - not experts in language design
  - combinations of extensions must "just work"

Challenges for composable language extensions

1. syntax — context free grammars, assoc. regexs
   - context-aware scanning [GPCE'07]
   - modular determinism analysis [PLDI'09]
   - Copper
2. semantics — attribute grammars
   - attribute grammars with forwarding, collections and higher-order attributes
   - set union of specification components
     - sets of productions, non-terminals, attributes
     - sets of attribute defining equations, on a production
     - sets of equations contributing values to a single attribute
   - modular well-definedness analysis [SLE'12a]
   - modular termination analysis [SLE'12b, Krishnan-PhD]
   - Silver

Building a parser from composed specifications.

$$\ldots \; CFG^{H} \cup^{*} \{CFG^{E_1}, \ldots, CFG^{E_n}\}$$

$$\begin{aligned} &\forall i \in [1, n].\mathit{isComposable}(CFG^{H}, CFG^{E_i}) \wedge \\ &\qquad\qquad \mathit{conflictFree}(CFG^{H} \cup CFG^{E_i}) \\ \Rightarrow \quad &\mathit{conflictFree}(CFG^{H} \cup \{CFG^{E_1}, \ldots, CFG^{E_n}\}) \end{aligned}$$

- Monolithic analysis - not too hard, but not too useful.
- Modular analysis - harder, but required [PLDI'09].
- Non-commutative composition of restricted LALR(1) grammars.

Building an attribute grammar evaluator from composed specifications.

$$\ldots\ AG^H \cup^* \{AG^{E_1}, \ldots, AG^{E_n}\}$$

$$\forall i \in [1, n].modComplete(AG^H, AG^{E_i})$$
$$\Rightarrow \quad complete(AG^H \cup \{AG_1^E, \ldots, AG_n^E\})$$

- Monolithic analysis - not too hard, but not too useful.
- Modular analysis - harder, but required [SLE'12a].

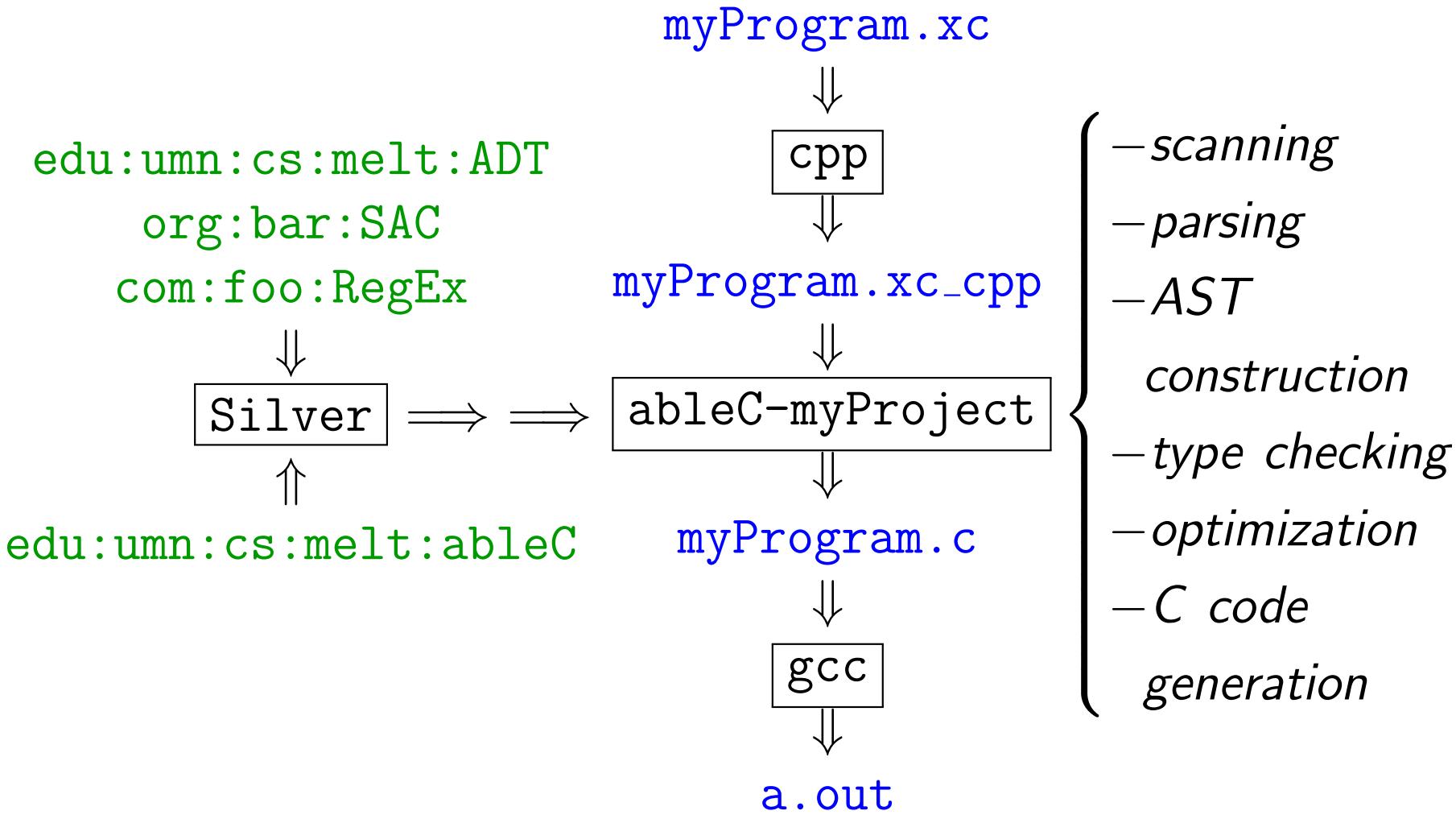
ableC:

Attribute grammar-Based Language Extensions for C

putting these ideas into practice

ableC: a specification of ISO C11 in Silver/Copper.

ableC


myProgram.xc
cpp
myProgram.xc_cpp
ableC-myProject
myProgram.c
gcc
a.out
edu:umn:cs:melt:ADT
org:bar:SAC
com:foo:RegEx
Silver
edu:umn:cs:melt:ableC
– scanning
– parsing
– AST construction
– type checking
– optimization
– C code generation

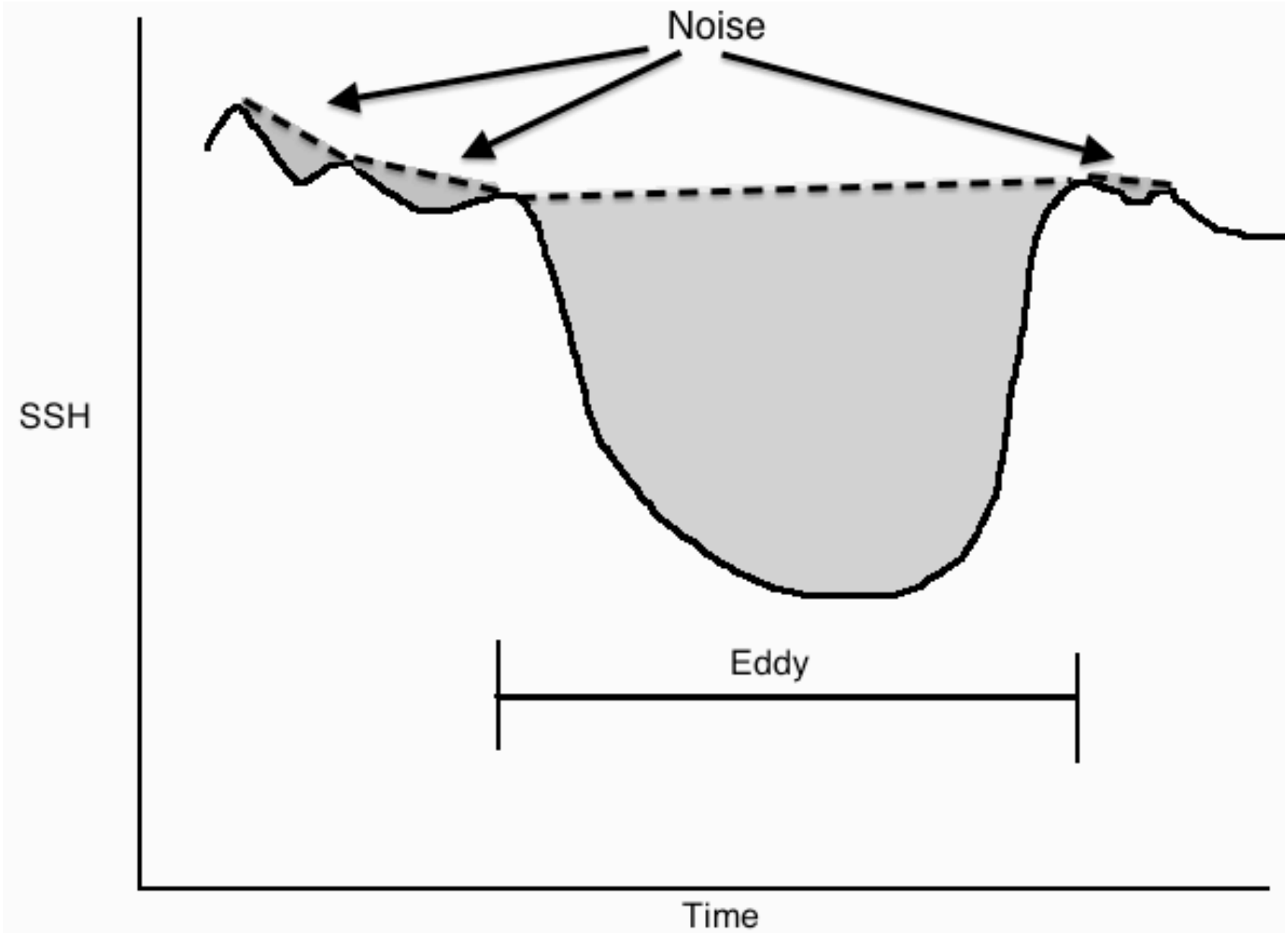
Sample language extensions

An agnostic's view.

Regular expressions

```
 1#include "stdio.h"
 2#include "regex.h"
 3
 4int main (int argc, char *argv[]) {
 5
 6   char *text = readFileContents("X.data") ;
 7
 8   // eukaryotic messenger RNA sequences
 9   regex foo = /^ATG[ATGC]{3,10}A{5,10}$/ ;
10
11   if ( text =~ foo )
12      printf ("Matches...\n") ;
13   else
14      printf ("Doesn't match...\n") ;
15}
```

A time slice for a point in the ocean



```
1 Matrix float<1> computeArea
2    (Matrix float<1> areaOfInterest)
3{
4   float y1 = areaOfInterest[0];
5   float y2 = areaOfInterest[end];
6   int x1 = 0;
7   int x2=dimSize(areaOfInterest,0)-1;
8   float m = (y1-y2) / ((float)(x1-x2));
9   float b = y1 - m*x1;
10  Matrix float<1> line = (x1::x2)*m+b;
11  float area
12    = with( x1 <= i < x2)
13        fold(+, 0.0, line - areaOfInterest);
14  return
15    with( 0 <= i < dimSize(line,0) )
16      genarray([dimSize(line, 0)], area);
17}
```

Halide-inspired transformations

- Halide: a C++ framework that gives programmers constructs to manipulate nested for-loops for image processing applications.
- It assumes that the C++ compiler will not generate code with high enough performance.
- The compiler can perform sophisticated optimizations, but does know now how to optimally apply them.
- Not for those seeking “casual parallelism”

```
1#include <stdlib.h>
2
3int main() {
4
5   int* grad =
6     (int*) malloc(100*100 * sizeof(int));
7
8   transform grad(x(0->100),y(0->100)) = x + y
9    with { split x by 4, xo, xi;
10          split y by 4, yo, yi;
11          reorder xo, yo, xi, yi
12        }
13
14  // best paired with an extension
15  // for 'proper' matrices
16}
```

```
1#include <stdlib.h>
2
3int main() {
4
5  int* grad =
6    (int*) malloc(100*100*100 * sizeof(int));
7
8  transform grad(x(0->100),y(0->100),z(0->100))
9    = x + y + z
10    with { split x by 4, xo, xi;
11           split y by 4, yo, yi;
12           split z by 4, zo, zi;
13           reorder xo, yo, zo, xi, yi, zi
14         }
15}
```

```
1#include <stdlib.h>
2
3int main() {
4
5   int* grad =
6     (int*) malloc(100*100 * sizeof(int));
7
8   transform grad(x(0->100),y(0->100)) = x + y
9     with { split x by 4, xo, xi;
10           split y by 4, yo, yi;
11           reorder xo, yo, xi, yi;
12           parallelize xo
13         }
14}
```

```
1#include <stdlib.h>
2
3int main() {
4
5   int *grad =
6     (int*) malloc(100*100 * sizeof(int));
7
8   transform grad(x(0->100),y(0->100)) = x + y
9     with { tile x, y, xo, yo, xi, yi, 4, 4 }
10}
```

Transformation extension

- These transformations generate the hard-to-write but expected nest of for loops.
- a reasonable intermediate language for high performance DSL-extensions
- Halide includes constructs for GPGPU targets, handling ghost cells and storing intermediate results for reuse
  all things we could add here

ADTs

- Implementing algebraic data types in C is sub-optimal.
  - syntactically verbose
  - error prone
  - unsafe
- A nice opportunity for a language extension.

```
1 /* Simple expresssion evaluation
2    using algebraic datatype as in
3    ML or Haskell. */
4
5 #include <stdio.h>
6 #include <stdlib.h>
7
8 typedef  datatype Expr  Expr;
9
10 datatype Expr {
11   Add (Expr*, Expr*);
12   Sub (Expr*, Expr*);
13   Mul (Expr*, Expr*);
14   Div (Expr*, Expr*);
15   Const (int);
16 };
17
```

```
18 int value (Expr *e) {
19   int res;
20   match (e) {
21     Add(e1,e2): res = value(e1) + value(e2);
22     Sub(e1,e2): res = value(e1) - value(e2);
23     Mul(e1,e2): res = value(e1) * value(e2);
24     Div(e1,e2): res = value(e1) / value(e2);
25     Const(v):   res = v;
26   };
27   return res;
28 }
29 int main () {
30   Expr *t = Add( Const(3),
31                  Mul(Const(2), Const(4)) ) ;
32   printf("value is %d\n", value(t) );
33   return 0;
34 }
```

ADT extension

- This extension generates the expected collection of
  - `struct` and `union` data structures.
  - constructor functions
- Coming soon
  - semantic analysis
  - disallowing access to underlying representation
  - pattern matching completeness checking

Questions.

ableC: a platform for DSL-feature implementation and distribution.

- What extensions should we build?
- What extensions might you like to build?
  We aren't domain experts in all of these areas!
- Are the restrictions too onerous?
- What can't you do?
  - language "modifications"
  - *e.g.* adding garbage collection for existing pointers

Questions?

Thanks for your attention.

`http://melt.cs.umn.edu`
`evw@cs.umn.edu`


Eric Van Wyk and August Schwerdfeger.
Context-aware scanning for parsing extensible languages.
In *Intl. Conf. on Generative Programming and Component Engineering, (GPCE)*, pages 63–72. ACM, 2007.


August Schwerdfeger and Eric Van Wyk.
Verifiable composition of deterministic grammars.
In *Proc. of Conf. on Programming Language Design and Implementation (PLDI)*, pages 199–210. ACM, June 2009.


Ted Kaminski and Eric Van Wyk.
Modular well-definedness analysis for attribute grammars.
In *Proc. of Intl. Conf. on Software Language Engineering (SLE)*, volume 7745 of *LNCS*, pages 352–371. Springer-Verlag, September 2012.


Lijesh Krishnan and Eric Van Wyk.
Termination analysis for higher-order attribute grammars.
In *Proceedings of the 5th International Conference on Software Language Engineering (SLE 2012)*, volume 7745 of *LNCS*, pages 44–63. Springer-Verlag, September 2012.


Lijesh Krishnan.
Composable Semantics Using Higher-Order Attribute Grammars.
PhD thesis, University of Minnesota, Department of Computer Science and Engineering, 2012.
`http://purl.umn.edu/144010`