Language extensions for parallel programming: opportunities and challenges

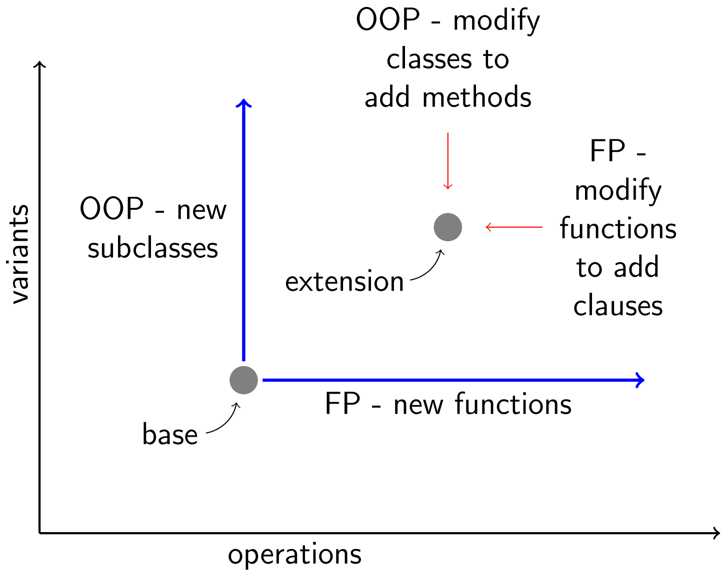
Eric Van Wyk

University of Minnesota

Bloomington, WG2.11, August 22-25, 2016

Language extensions and the expression problem

Directions of Extensibility



The Expression Problem

Requirements for solving it:

1. extensibility in both directions
2. strong static typing
3. no modification of existing code
4. separate compilation and type checking

Old problem, new name popularized by Phil Wadler.

Allows

1. a linear ordering of extensions
   $Base \lhd E_1 \lhd E_2$
2. E_2 developer writes code to handle E_1

Independently extensible version

Requirements for solving it:

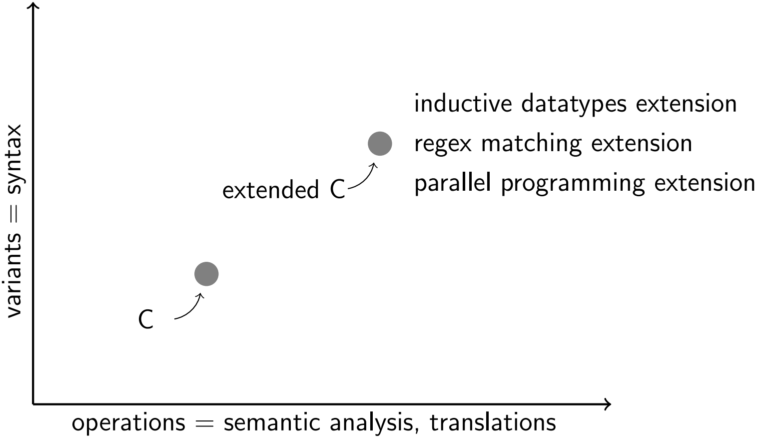
1. extensibility in both directions
2. strong static typing
3. no modification of existing code
4. separate compilation and type checking
5. no linear ordering of extensions, Zenger and Odersky
   $Base \lhd E_1 \lhd E_2$
   $Base \lhd E_2 \lhd E_1$

Allows

1. "glue" code to compose extensions
   written by 3^{rd} party to compose E_1 and E_2
   e.g. the operation in E_1 for variant in E_2

Extensible Languages

Base = Host Programming Language



```
typedef datatype Tree Tree;
datatype Tree {
  Fork ( Tree*, Tree*, const char* );
  Leaf ( const char* );
};
cilk int count_matches (Tree *t) {
  match ( t ) {
    Fork(t1,t2,str): {
      int res_t, res_t1, res_t2;
      spawn res_t1 = count_matches( t1 );
      spawn res_t2 = count_matches( t2 );
      res_t = ( str =~ /foo[1-9]+/ ) ? 1 : 0;
      sync;
      cilk return res_t1 + res_t2 + res_t ;
    } ;
    Leaf(/foo[1-9]+/): { cilk return 1 ; } ;
    Leaf(_): { cilk return 0 ; } ;
    } ;
  }
```

Another Expression Problem

Requirements for solving it:

1. extensibility in both directions
2. strong static typing
3. no modification of existing code
4. separate compilation and type checking
5. no linear ordering of extensions
   $Host \lhd E_1 \lhd E_2$
   $Host \lhd E_2 \lhd E_1$
6. no glue code, composition is automatic

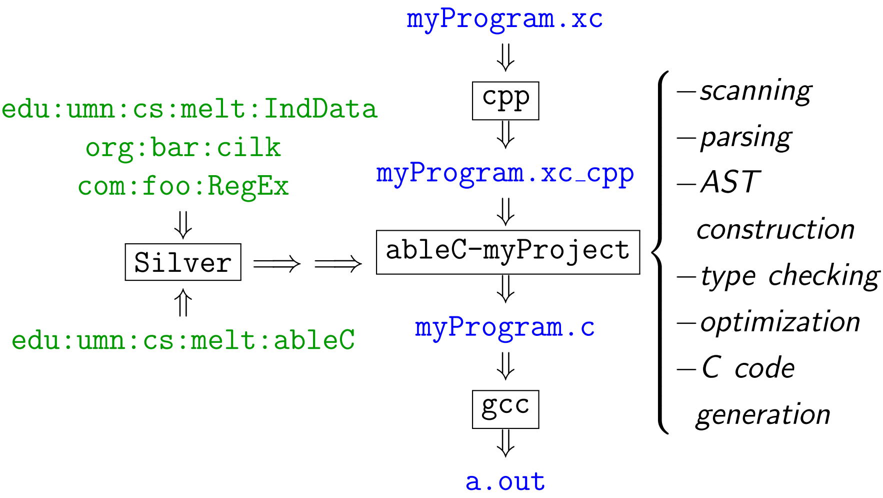
Allows

- a non-expert programmer to do the composition

But it requires

- extensions are somehow realizable in the base

ableC- extensible specification of C11



Another Expression Problem

How we solve it for extensible languages:

1. extensibility in both directions — Attribute grammars
2. strong static typing — Effective completeness analysis
3. no modification of existing code — Attribute grammars
4. separate compilation and type checking — Modular effective completeness analysis
5. no linear ordering of extensions — Attribute grammars
   $Host \lhd E_1 \lhd E_2$
   $Host \lhd E_2 \lhd E_1$
6. no glue code, composition is automatic — Forwarding

Requires

- extension language constructs translate down to host language constructs

Why language extensions for parallel programming?

Programmer's perspective

A great deal of diversity in linguistic abstractions for parallel programming.

No "right" set of abstractions for parallel programming.

What is right depends on many factors:

- the application or problem at hand,
- sophistication and personal preferences of the programmer,
- the degree of performance desired and effort required to achieve it.
- ...

Choosing abstractions has a high up front cost

- Adopting a new language - high up front cost - hard to experiment in one's current project.

  Switching a project to, say X10, is expensive.

- Also prevents one from using different forms of parallelism in different parts of the same program.

Researcher's perspective

Consider Cilk.

- Researchers built a new C source-to-source translator: parsing and semantic analysis. This is a lot of work.
- Long journey from research compiler into Intel's C compilers.
  A trip that is rarely repeated.

Parallel programming may be a "killer app" for extensible languages.

Cilk as a language extension

The MIT implementation of Cilk

Two significant components:

- a Cilk-to-C translator
- a sophisticated task-based run-time implementing efficient work-stealing scheduler, written in C.

A language extension replaces the translator,
but not the run-time.

The ableC implementation

- Parsing, simple.
- Semantic analysis
  - A `cilk return` used in `cilk` function.
  - `spawn` calls a `cilk` function.
- Code generation
  - create a fast and slow "clone" for each `cilk` function.
  - requires handling, *e.g.*, the `match` statement

Local transformations

Extension syntax locally expands (forwards) to its translation to C.[1]

- MIT Cilk:
  ```
  res_t1 = spawn count_matches( t1 );
  ```
- ableC Cilk:
  ```
  spawn res_t1 = count_matches( t1 );
  ```

Code generated uses `res_t1` in a few places and thus needs to be part of the extension.

[1] Lifting of new declarations is supported.

Adding, not changing behavior

Extensions can add to, but not change, behavior of existing host language constructs.

- MIT Cilk:
  `return res_t1 + res_t2 + res_t ;`

- ableC Cilk:
  `cilk return res_t1 + res_t2 + res_t ;`

Code generated for `cilk return` is non-trivial in one of the clones.

A non-cilk `return` in a cilk function does raise an error.

Composition vs Expressiveness

- Guarantees of composability impose some restrictions.
- These previous issues are concerns for any extension.

Challenges

Multiple parallel programming extensions

Compute the sum of the squares of numbers stored in a tree using 3 forms of parallelism.

```
typedef datatype Tree Tree;

datatype Tree {
    Fork (Tree*, Tree*, const int, const float*);
    Leaf (const int, const float*);
};
```

```
int square (int x) { return x * x; }

cilk int treeSumOfSqs (Tree *t) {
  match (t) {
    Fork(t1, t2, size, values): {
      int t1res, rest2, lsos;
      spawn t1res = treeSumOfSqs(t1);
      spawn t2res = treeSumOfSqs(t2);
      spawn lsos = sumOfQuares(size, values);
      sync;
      cilk return t1res + t2res + lsos;
    }
    Leaf(size, values): {
      cilk return
        fold ( (+), 0.0, size,
          map ( square, size, values) );
    } ;
  }
}
```

```
cilk int sumOfSquares (const int size,
                       const float *values) {
  int sos = 0;
  transform {
    for (int i=0; i<size; ++i)
      sos = sos + square(values[i]);
  } by split i by 4 into i_in, i_out,
       vectorize i_in;
  cilk return sos;
}
```

Static interaction

- Don't parallelize inner loops.
- Similarly, perhaps "inner" parallel constructs should generate sequential code.

  The parallel map/fold used in leaves of the Cilk tasks should perhaps just be executed sequentially.
- What is the static protocol through which independently-developed extensions communicate?
- The symbol table through which extensions may communicate is an example of a static protocol.

Dynamic interaction

Parallel run-times manage and schedule resources

- Different extension run-times may conflict
- over schedule resources
- result in low performance

Is there some common run-time applicable to many abstractions?

Next steps

- More systematic implementation of various approaches to parallel programming as language extensions.
- What parallel programming features should we implement?
- Yours?
- What can't work here?
  Negative examples are important here.
- Collaboration opportunities.

Thanks for your attention.