

Imperative, Declarative, Functional and Domain-Specific Programming... Oh My!



Patrick McCormick

March 25, 2016

SOS 20 Workshop

March 22 - 25, 2016

Asheville, North Carolina



Are we entering a new age of software development for HPC?

Yes... I hope so.... Would like to think so...

But we still have a ways to go...

The “Simple” set of Goals

- **Performance (fast)**
- **Portability (run everywhere)**
 - Fast... Standardized...
- **Productivity/Programmability**
 - Easily, everywhere, and high-performance
- **What we are typically asking for is sequential semantics with parallel execution...**

Is the key Cost Really Data Movement?

- *“Data movement is expensive, compute is free.”*
- **But...**
 - *Idle processors are not free*
 - **Trinity:** *If you dump data from memory to disk you spend 10X more power waiting on the data to move than to move the data!*
- **So, no surprise, we really want to keep processors busy...**

Operation	Energy (pJ)
64-bit integer operation	1
64-bit floating-point operation	20
256 bit on-die SRAM access	50
256 bit bus transfer (short)	26
256 bit bus transfer (1/2 die)	256
Off-die link (efficient)	500
256 bit bus transfer (across die)	1,000
DRAM read/write (512 bits)	16,000
HDD read/write	$O(10^6)$

Courtesy Greg Asfalk (HP) and Bill Dally (NVIDIA)

The Importance of Programming Abstractions

Imperative, explicit data movement:

- *Focus on control flow, explicit parallelism and low-level data abstractions*

```
AsyncRecv ( X ) ;  
DoWork ( Y ) ;  
Sync ( ) ;  
F ( X ) ;
```

- *How much work should I do?*
- *Is this performance portable?*
- *When does forward progress really occur?*
- *What if I have more work and data movement happening in DoWork?*
 - *What resources are in use?
Where is the data? Who is using it and how?*
- *Is this modular?*

Concept from: Mike Bauer's Thesis (Stanford),
[*Legion: Programming Distributed Heterogeneous Architectures with Logical Regions*](#)

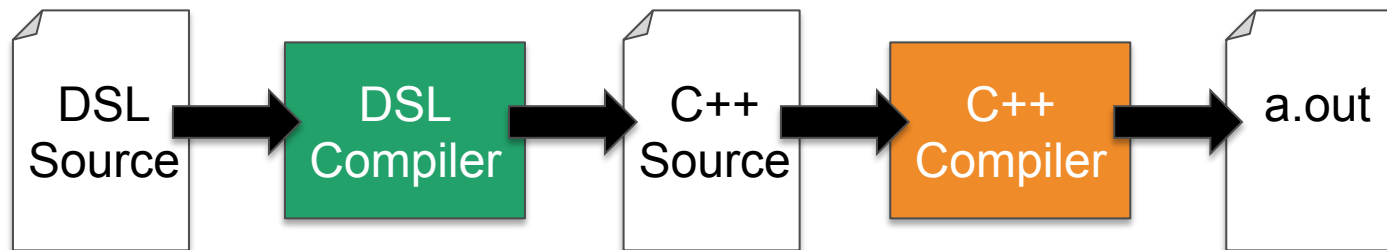
Simplifying the Challenge

“Domain-Specific” Languages

- **Why?**
 - Improved productivity, better maintainability, portability, validation, improved optimizations, thus improved reliability and performance
- **But...**
 - Risks in terms of costs associated with their design, implementation, adoption, maintenance/longevity, and education...
- **What can be done to reduce the risks/costs?**

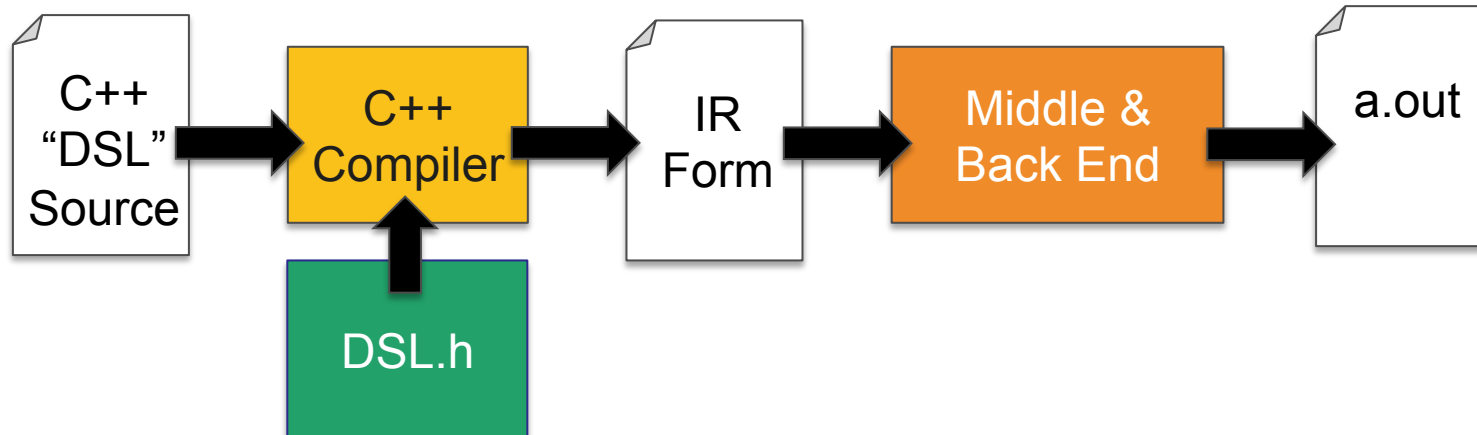


“Standalone” Source-to-Source Compilation



- **Simplified compiler – heavy lifting done by “real” compiler**
 - Great way to prototype...
- **But... Custom language maintenance issues**
- **Domain knowledge/semantics lost in code generation...**
 - DSL compiler optimizations can be undone by C++ compiler
 - Developer ends up with C++ tools...

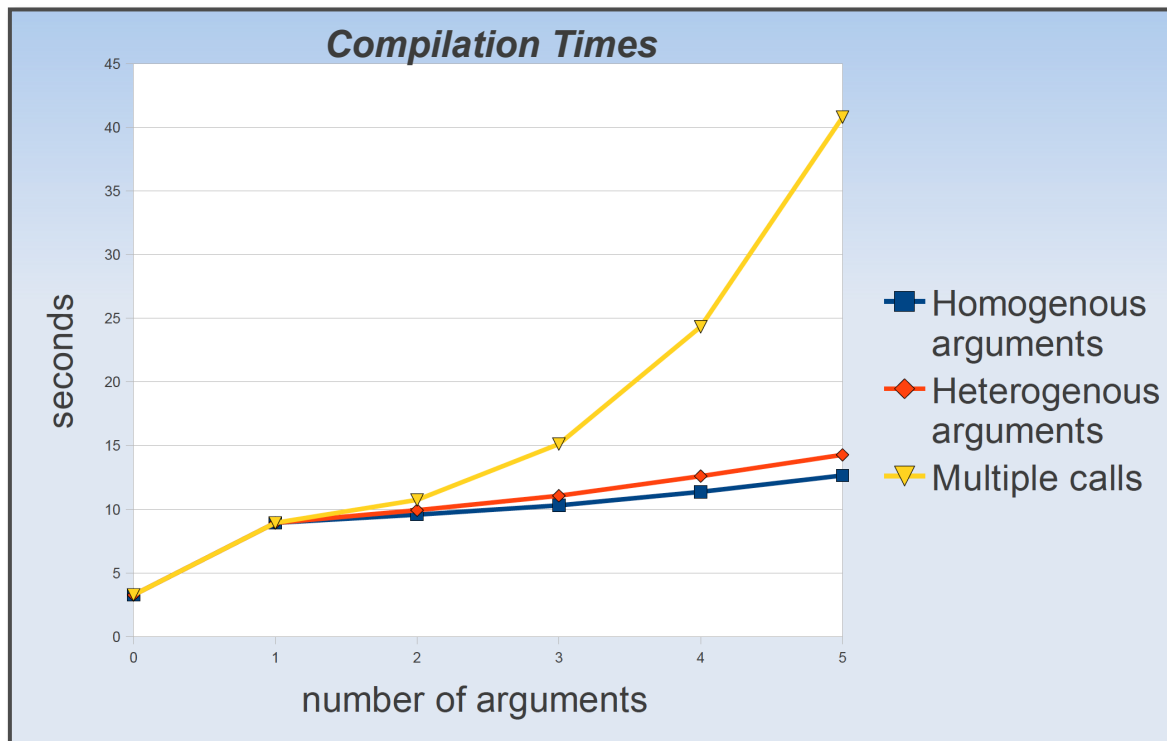
C++ Embedded DSLs



- **Meta-programmed code generation - standardized “goodness”**
- **Underlying infrastructure can be complex and difficult (and often not as opaque as we might like). Stuck w/ C++ semantics and syntax...**
- **Once again, domain knowledge/semantics lost in code generation (after template expansion)...**
 - Can be hard to optimize, match semantic goals due to host language restrictions
 - Developer (and optimizer) ends up with expanded “goop”

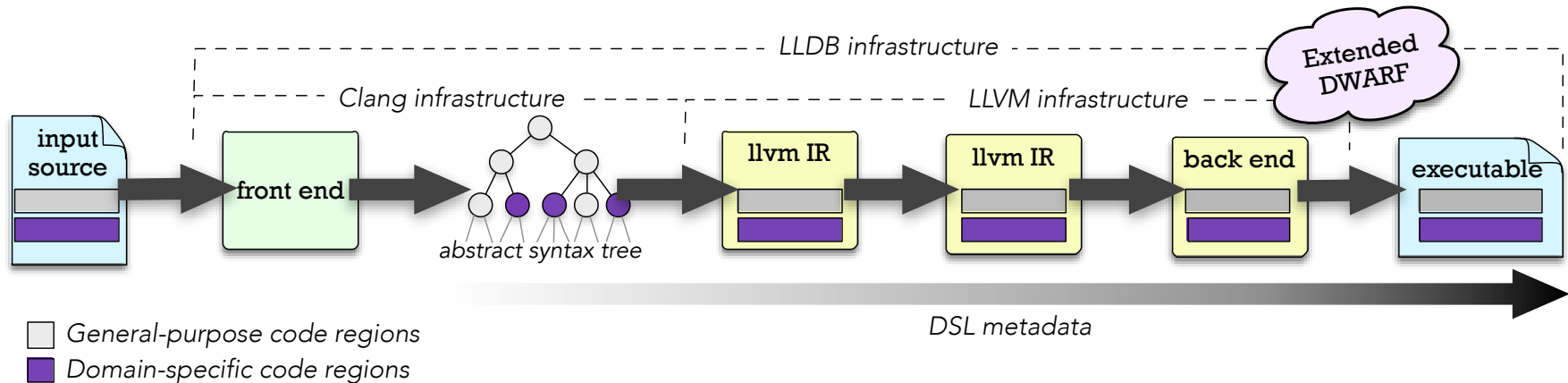
Can you Spare a Minute? More C++ EDSL worries...

```
safe::printf<_S("Hello %s!")>("World!");
```



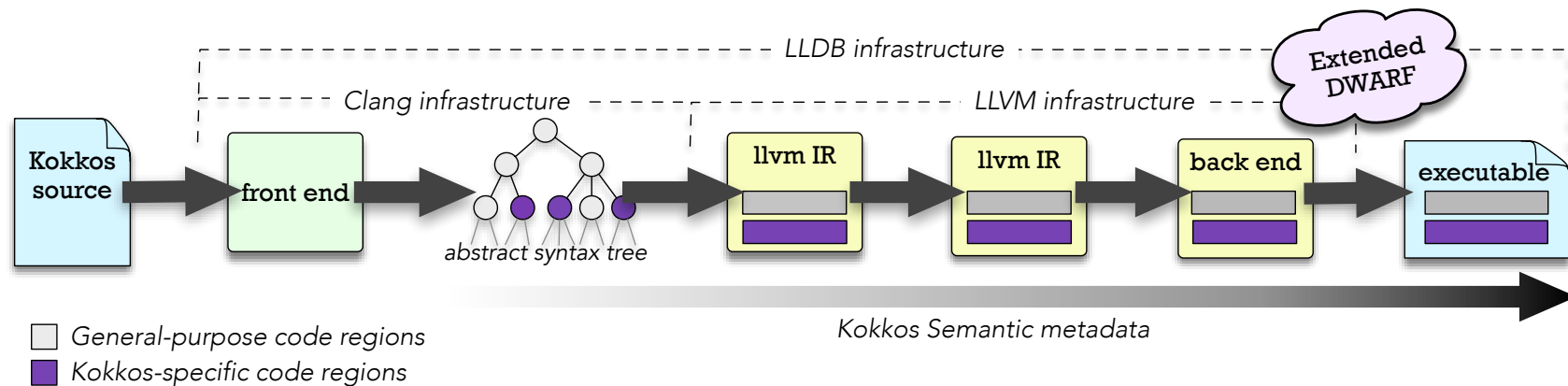
"Domain-specific Language Integration with Compile-time Parser Generator Library", Zoltan Porkolab and Abel Sinkovics, Proceeding GPCE '10 the ninth international conference on Generative Programming and Component Engineering.

Domain-Aware Toolchains



- **We really want a fully supported toolchain – not just a set of “front end” semantics and abstractions...**
 - Allow the developer (and the compiler) to reason in terms of the original abstractions (not the “goop”)
- **OpenMP implementations do/can have similar issues...**

Kokkos-Aware Clang



- **Code generation phase of Clang intercepts Kokkos constructs prior to template expansion and implements semantics-aware code generation**

- SC15 tutorial code:

- Compile time is approx. 4.5 times faster
- Code generation: parallel-for about 5% faster (GPUs), reductions need to be optimized (about 2-3x slower at present)



Due to Complexity we Often Only Look at one Piece of the Puzzle...

- **How do we?**
 - Interoperate across different models/abstractions, languages and legacy code bases?
 - Build a set of useful and flexible tools for understanding details in terms of the abstractions we're developing with?
 - Get applications to adopt new approaches for programming?