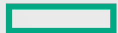
HPE

Evolving a Research Prototype: Dyno, Chapel’s New Compiler Front-End

Daniel Fedorin, HPE

March 18, 2026

The State of Things



Selection Pressure

- A brand-new programming language promising productivity gains must live up to its promise
  - You say “I can make a performant high-level parallel programming language”, so do it!
  - You must prove your worth against established technologies
  - This happens in a very direct and real way when funded by a research program

- Part of the work was exploratory
  - “Can we find a way to make this pattern easy to write and performant to run?”

- To accelerate pace, new, high-level features could be implemented in terms of low-level ones
  - Decomposing features in this way also helps ensure a small, focused, and understandable core

The Resulting Compiler

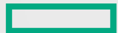
- Uses whole-program compilation
  - This leads to a simpler compiler architecture

- Uses a very limited set of constructs to which all programs boil down to
  - Variable definitions, blocks and calls are much of what the compiler reasons about
  - This is a low-level encoding

- Uses global, mutable state
  - If something can be optimized out, remove it from the tree!
  - If a user-written program implicitly calls a procedure, add an explicit call into the representation!
  - This is still used by compilers today (e.g., MLIR's progressive lowering), but it's not suitable for front-ends

- Mixes concerns
  - Lowering, optimization and type resolution are all intertwined
  - You can't ask about the type of a variable without *also* doing "normalization" into SSA, e.g.

Towards Modernity



New Goals

- As proofs of concepts give way to big projects, the compiler needs to scale
  - Whole-program compilation on big projects means restarting from scratch every time
  - Most of the code in a large project doesn’t change between compilations
  - Separate compilation is one solution

- Users expect much more comfort than a command-line compiler
  - Diagnostics in modern languages have made great strides in recent-ish years
  - In particular, editor tools are often interactive and must respond to very quick (but small) changes

- Ideas accrued from the exploration now can be reconciled and unified
  - A reimplementation can learn from the mistakes

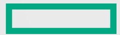
Compiler as a Library

- Some tools we might want include:
  - Documentation generator (chpldoc)
  - Editor-integration (Language Server Protocol server)
  - Ad-hoc programs to analyze and reason about code

- To make the compiler do the work of other tools will complicate matters.
  - Each tool has different concerns
  - Each tool’s interaction with the command-line or file system is different
  - All the code would have to co-exist in the compiler

- Instead, can we make the compiler into a library, and build other tools on top?
  - Yep, and that’s what we did

Dyno:
Front-End As a Library



Dyno: Front-End As a Library

The idea with Dyno was to produce a new implementation of the compiler front-end, handling:

- Parsing
- Scope Resolution (mapping names to what they refer to)
- Type Resolution (figuring out types of expressions, resolving function overloads, etc.)

Dyno ought to be usable as a library, so that other tools can leverage it for various purposes.

Shortcuts You Can't Take

When turning the compiler into a library, you lose the ability to do some things:

- **Exiting on Error:** no matter how ir… is, the compiler must keep going.
- **Printing Errors:** for proper editor i… sed along as data, not dumped
- **Global State:** a user can easily op… time
- **Mutable State:** if you modify som… there before
- **Immediate Lowering:** you can de… t tools would have to piece it back

Errors to Aspire To

```
-- TYPE MISMATCH ---------------------------------------- types/missing-field.elm

The 1st argument to function `isOver50` has an unexpected type.

13|   isOver50 hermann
               ^^^^^^^
Looks like a record is missing the field `age`

As I infer the type of values flowing through your program, I see a conflict
between these two types:

    { a | age : comparable }

    { first : String, last : String }
```

Evan Czaplicki,
[Compiler Errors for Humans]

```
error[E0382]: borrow of moved value: `s`
 --> <anon>:9:20
  |
6 |     let s = S(0);
  |         - move occurs because `s` has type `S`, which does not implement the
`Copy` trait
7 |     let s2 = s;
  |              - value moved here
8 |
9 |     println!("{}", s.0);
  |                    ^^^ value borrowed here after move
  |
  = note: this error originates in the macro `$crate::format_args_nl` (in
Nightly builds, run with -Z macro-backtrace for more info)
```

Kobzol's blog,
[Evolution of Rust compiler errors]

Explicit Errors

- If we're making errors into objects to hand of to tools, we need to think about what data is bundled in each of them.
- For nice error messages like in Elm or Rust, we want different data for different errors.
- To make this type-safe and convenient, we need to introduce different representations depending on the exact error type.
- This also lets us think about how to print errors on a case-by-case basis

```
ERROR_CLASS(AmbiguousConfigSet)
ERROR_CLASS(AmbiguousIdentifier)
ERROR_CLASS(AmbiguousVisibilityIdentifier)
ERROR_CLASS(AsWithUseExcept)
ERROR_CLASS(AssignFieldBeforeInit)
ERROR_CLASS(ConstRefCoercion)
WARNING_CLASS(DeprecatedSyncRead)
WARNING_CLASS(Deprecation)
ERROR_CLASS(DotExprInUseImport)
ERROR_CLASS(DotTypeOnType)
ERROR_CLASS(EnumAbstract)
ERROR_CLASS(EnumInitializerNotParam)
ERROR_CLASS(EnumInitializerNotInteger)
ERROR_CLASS(EnumValueAbstract)
ERROR_CLASS(ExternCCompilation)
WARNING_CLASS(GenericFieldWithoutMark)
ERROR_CLASS(IfVarNonClassType)
WARNING_CLASS(ImplicitFileModule)
ERROR_CLASS(IncompatibleIfBranches)
ERROR_CLASS(IncompatibleKinds)
ERROR_CLASS(IncompatibleRangeBounds)
ERROR_CLASS(IncompatibleTypeAndInit)
ERROR_CLASS(IncompatibleYieldTypes)
ERROR_CLASS(InterfaceAmbiguousFn)
ERROR_CLASS(InterfaceInvalidIntent)
ERROR_CLASS(InterfaceMissingAssociatedType)
ERROR_CLASS(InterfaceMissingFn)
ERROR_CLASS(InterfaceMultipleImplements)
ERROR_CLASS(InterfaceNaryInInherits)
ERROR_CLASS(InterfaceReorderedFnFormals)
ERROR_CLASS(InvalidClassCast)
ERROR_CLASS(InvalidContinueBreakTarget)
```

* The (abridged) code on the right uses a C++ pattern called X-macros to code-generate much boilerplate.

Chapel’s Errors

- The results of these changes are available in the production compiler.
- The code below demonstrates an error that today’s Chapel compiler can produce.
  - We still have a “brief” mode to keep errors concise and information-dense.

```
── error in fullUsePath.chpl:9 [UseImportUnknownMod] ──
  Cannot find module or enum named 'ToLookFor'.
  In the following 'use' statement:
      |
    9 |                 use ToLookFor;
      |
      |
  The following declarations are not covered by the 'use' statement but seem similar to what you meant.

  A declaration of 'ToLookFor' is here:
      |
    1 | param ToLookFor = "??";
      |
      |
  However, 'use' statements can only be used with modules or enums (and this 'ToLookFor' is not a module or enum).
```

Explicit AST

- Instead of the small set of constructs into which everything is lowered, we now have a representation that matches the source code closely.
- Syntax-driven analyses can now be written very directly
- No ”pattern matching” on the desugared representation is needed

```
def check(node):
    if isinstance(node, Loop):
        print("It's a loop!")
    else:
        print("It's not a loop...")
```

* This is Python code. More on this later

```
AST_BEGIN_SUBCLASSES(Call)
  AST_NODE(FnCall)
  AST_NODE(OpCall)
  AST_NODE(PrimCall)
  AST_NODE(Reduce)
  AST_NODE(Scan)
  AST_NODE(Tuple)
  AST_NODE(Zip)
AST_END_SUBCLASSES(Call)

AST_BEGIN_SUBCLASSES(Decl)
  AST_NODE(MultiDecl)
  AST_NODE(TupleDecl)
  AST_NODE(ForwardingDecl)

  AST_BEGIN_SUBCLASSES(NamedDecl)
    AST_NODE(EnumElement)
    AST_NODE(Function)
    AST_NODE(Interface)
    AST_NODE(Module)
    AST_NODE(TypeQuery)
    AST_NODE(ReduceIntent)
```

Motivating Query-Based Compilers

- The expected usage from a text editor is often very different from usage from the command-line
- Users will make many small changes quickly, and likely want updated information interactively
  - Or, at the very least, on 'save'
- If you naively re-resolve each file on edit, you do a lot of work over and over again.

- Example usage scenario:
  - Open a 1000 line source file
  - Find an expression, change it from 'x' to 'x+1'
  - Realize that it ought to be 'x+2', change it again
  - Most of the file hasn't changed!

- Even when looking at a single file, we want something to avoid redundant work

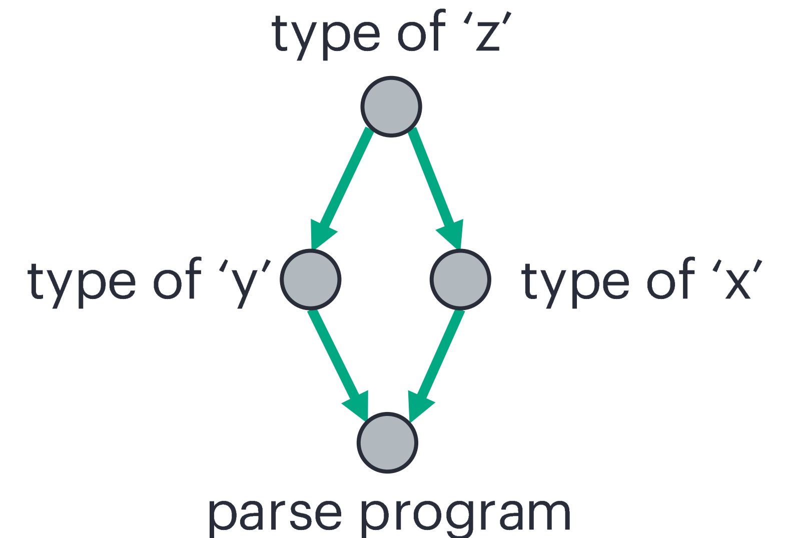
Query-Based Compilers

- A *query-based compiler* is based around "questions about a codebase" (queries)
- Some queries are: "what's the AST of this file?", or "What's the type of X?"
- The compiler ought to be able to answer each query independently
  - In any order
  - While doing only the work needed for that query
- Libraries can ask (query) anything about a codebase without understanding the compiler's pipeline
- Query results should be re-usable
  - This means they can't rely on global state or have side effects (e.g., mutable variables)

Worked Example

- Query: “What’s the type of ’x’?”
  - Parse the program, resolve ‘x’
- Query: “What’s the type of ‘y’?”
  - Already parsed program, now resolve ‘y’
- Query: “What’s the AST of the program”?
  - Already parsed program, return result

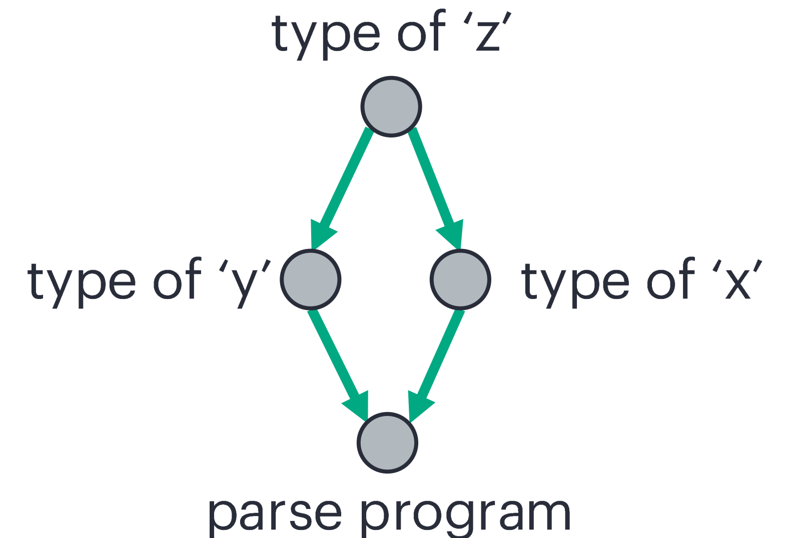
```
var x = 1;
var y = 2;
var z = x+y;
```



Worked Example

- Query: “What’s the type of ’z’?”
  - Parse the program, resolve ‘x’, ‘y’, ‘z’
- Change ’x’ to be ‘42’
  - This makes our information stale.
- Query: “What’s the type of ’z’?”
  - Re-parse the program
  - AST of ‘y’ hasn’t changed, no need to re-compute
  - AST of ‘x’ has changed, recompute ‘x’
  - Types of ‘x’ and ‘y’ haven’t change, no need to re-compute type of ‘z’

```
var x = 1;
var y = 2;
var z = x+y;
```

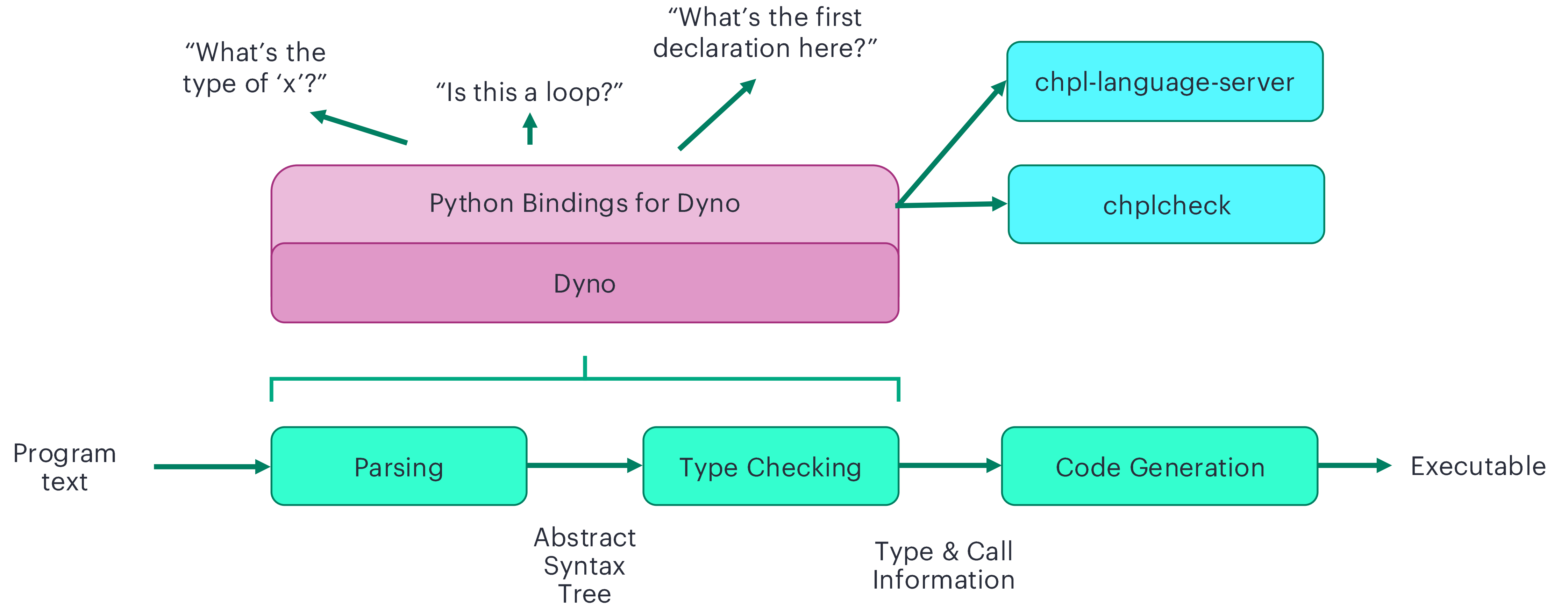


Separate Compilation in a Whole-Program Language

- We talked about separate compilation, but so far, we only have incrementality
- Chapel's semantics are not aware of separate compilation — they assume whole-program
- However, by saving results of queries, we can achieve a similar effect
  - A single unit isn't a 'compiled object file' but a bundle of 'all the queries about this file so far'
  - Loading the queries means not having to re-compute them from scratch
  - Creating the query file file can be done separately

Python Bindings

By making the compiler a library, we can pipe (with some effort) pipe its features to a high(er)-level language!



Language Server

```
1   use BlockDist;
2   config const n = 100_000,
3                x = 1,
4                y = 3;
5   const D = {1..n, 1..n};
6   var A = Block.createArray(D, int);    Warning: [Deprecation]: 'Block' is deprecated, please use 'blockDist' instead
7   forall a in A {
8     if x < here.id < y {    Error: [NonAssociativeComparison]: comparison operators are not associative
9       a = 1;
10    } else {
11      a = 0;
12    }
13  }
```

```
// This procedure writes a square
proc writeSquareArray(n, X, filen

  // Create and open an output fi
  var outfile = open(filename, ioMode.cw);
  var writer = outfile.writer(locking=false);
```

```
enum ioMode

The :type: ioMode type is an enum. When used as arguments when opening f
fopen() in C. However, :proc: open() in Chapel does not necessarily invok
```

```
1   enum color {
2     red = 1,
3     green = 2,
4     blue = 3,
5     cyan = red : int + 10 /* 11 */,
6     magenta = 12,
7     yellow = 13
8   }
```

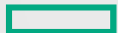
Linter

```
for (i, j) in zip(1..10, 1..10) {     Lint: rule [UnusedLoopIndex] violated
  writeln("i = ", i);
   write(" j = ", i);     Lint: rule [IncorrectIndentation] violated
}

proc saxpy(n:int, s:int, a: [], b: []) {     Lint: rule [UnusedFormal] violated
  var c: [0..#n] int;
  foreach i in {0..#n} {     Lint: rule [SimpleDomainAsRange] violated
```

Quick Fix
- Apply Fix for SimpleDomainAsRange
- Apply Fix for SimpleDomainAsRange (Ignore this warning)
- Fix using Copilot
- Explain using Copilot

Challenges



Query-Based Compilers + Chapel

“The effectiveness of query-based compiler is limited by the dependency structure of the source language. “

— Alex Kladov, Against Query Based Compilers

```
module ChapelStandard {
  public use ChapelBase;
```

```
module ChapelBase {
  public use ChapelStandard;
```

Working Around Recursion

- Infinite recursion is not allowed in general
  - The compiler had better run in finite time!
- Normally, we can avoid this with a stateful list of ‘visited’ / ‘seen’ inputs
  - We don’t recurse if we’ve already considered / are currently considered an input
- But that’s mutable state, so queries can’t do it!
- Some ways to work around this:
  - A single query that explicitly computes the dependency graph (using internal mutable state)
  - Meta-questions in queries: “is this query already running?”
    - This **is** is a side effect, and it **will** cause problems.

Other Challenges

- No good ways to save ‘auxiliary results’ computed as part of another query
  - In the incremental model, you need to know how to recompute just the data you need
  - You’d need to know that a different computation produced auxiliary results, and re-run THAT.
  - But that changes the dependency structure…
- Performance and memory overhead
  - Tracking dependency structure, saving results, seeing what changed etc. all takes extra work.
  - Some hot queries (eg, “find AST for variable”) can create a lot of entries and consume memory, slowing lookup
  - ~5%+ performance cost to dependency tracking alone
- Some ‘usual’ operations are unnatural
  - e.g., no notion of a ‘current call stack’ in queries, because that’s outer (possibly mutable) state
- Tracking (and re-emitting) errors when needed is nontrivial
  - An error should be emitted even if a query isn’t re-executed, and its result is re-used

Thank You

© 2026 Hewlett Packard Enterprise Development LP