Type-Level Programming in Chapel for Compile-Time Specialization

Daniel Fedorin, HPE

Compile-Time Programming in Chapel

- **Type variables**, as their name suggests, store types instead of values.

```
type myArgs = (int, real);
```

- **Procedures with `type` return intent** can construct new types.

```
proc toNilableIfClassType(type arg) type do
  if isNonNilableClassType(arg) then return arg?; else return arg;
```

- **`param` variables** store values that are known at compile-time.

```
param numberOfElements = 3;
var threeInts: numberOfElements * int;
```

- **Compile-time conditionals** are inlined at compile-time.

```
if false then somethingThatWontCompile();
```

Restrictions on Compile-Time Programming

- Compile-time operations do not have mutable state.
  - Cannot change values of `param` or `type` variables.
- Chapel's compile-time programming does not support loops.
  - `param` loops are kind of an exception, but are simply unrolled.
  - Without mutability, this unrolling doesn't give us much.
- Without state, our `type` and `param` functions are pure.

Did someone say "pure"?

I can think of another language that has pure functions...

- ✅ Haskell doesn't have mutable state by default.
- ✅ Haskell doesn't have imperative loops.
- ✅ Haskell functions are pure.

Programming in Haskell

Without mutability and loops, (purely) functional programmers use pattern-matching and recursion to express their algorithms.

- Data structures are defined by enumerating their possible cases. A list is either empty, or a head element followed by a tail list.

```haskell
data ListOfInts = Nil | Cons Int ListOfInts

-- [] = Nil
-- [1] = Cons 1 Nil
-- [1,2,3] = Cons 1 (Cons 2 (Cons 3 Nil))
```

- Pattern-matching is used to examine the cases of a data structure and act accordingly.

```haskell
sum :: ListOfInts -> Int
sum Nil = 0
sum (Cons i tail) = i + sum tail
```

Evaluating Haskell

Haskell simplifies calls to functions by picking the case based on the arguments.

```
sum (Cons 1 (Cons 2 (Cons 3 Nil)))

-- case: sum (Cons i tail) = i + sum tail
= 1 + sum (Cons 2 (Cons 3 Nil))

-- case: sum (Cons i tail) = i + sum tail
= 1 + (2 + sum (Cons 3 Nil))

-- case: sum (Cons i tail) = i + sum tail
= 1 + (2 + (3 + sum Nil))

-- case: sum Nil = 0
= 1 + (2 + (3 + 0))

= 6
```

A Familiar Pattern

Picking a case based on the arguments is very similar to Chapel's function overloading.

- **A very familiar example**:

```
proc foo(x: int) { writeln("int"); }
proc foo(x: real) { writeln("real"); }
foo(1);   // prints "int"
```

- **A slightly less familiar example**:

```
proc foo(type x: int) { compilerWarning("int"); }
proc foo(type x: real) { compilerWarning("real"); }
foo(int);   // compiler prints "int"
```

A Type-Level List

Hypothesis: we can use Chapel's function overloading and types to write functional-ish programs.

```
record Nil {}
record Cons { param head: int; type tail; }


type myList = Cons(1, Cons(2, Cons(3, Nil)));



proc sum(type x: Nil) param do return 0;
proc sum(type x: Cons(?i, ?tail)) param do return i + sum(tail);

compilerWarning(sum(myList) : string); // compiler prints 6
```

```
data ListOfInts = Nil
                | Cons Int ListOfInts


myList = Cons 1 (Cons 2 (Cons 3 Nil))


sum :: ListOfInts -> Int
sum Nil = 0
sum (Cons i tail) = i + sum tail
```

Type-Level Programming at Compile-Time

After resolution, our original program:

```
record Nil {}
record Cons { param head: int; type tail; }

type myList = Cons(1, Cons(2, Cons(3, Nil)));

proc sum(type x: Nil) param do return 0;
proc sum(type x: Cons(?i, ?tail)) param do return i + sum(tail);

writeln(sum(myList) : string); // compiler prints 6
```

Becomes:

```
writeln("6");
```

There is no runtime overhead!

Type-Level Programming at Compile-Time

Type-Level Programming at Compile-Time

> Why would I want to do this?!
>
> - You, probably

- Do you want to write parameterized code, without paying runtime overhead for the runtime parameters?
    - **Worked example**: linear multi-step method approximator
- Do you want to have powerful compile-time checks and constraints on your function types?
    - **Worked example**: type-safe `printf` function

Linear Multi-Step Method Approximator

Type-Safe `printf`

The `printf` Function

The `printf` function accepts a format string, followed by a variable number of arguments that should match:

```
// totally fine:
printf("Hello, %s! Your ChapelCon submission is #%d\n", "Daniel", 18);

// not good:
printf("Hello, %s! Your ChapelCon submission is #%d\n", 18, "Daniel");
```

Can we define a `printf` function in Chapel that is type-safe?

Yet Another Type-Level List

- The general idea for type-safe `printf`: take the format string, and extract a list of the expected argument types.
- To make for nicer error messages, include a human-readable description of each type in the list.
- I've found it more convenient to re-define lists for various problems when needed, rather than having a single canonical list definition.

```
record _nil {
  proc type length param do return 0;
}
record _cons {
  type expectedType;   // type of the argument to printf
  param name: string; // human-readable name of the type
  type rest;

  proc type length param do return 1 + rest.length();
}
```

Extracting Types from Format Strings

```
proc specifiers(param s: string, param i: int = 0) type {
  if i >= s.size then return _nil;

  if s[i] == "%" {
    if i + 1 >= s.size then
        compilerError("Invalid format string: untermintéd %");

    select s[i + 1] {
      when "%" do return specifiers(s, i + 2);
      when "s" do return _cons(string, "a string", specifiers(s, i + 2));
      when "i" do return _cons(int, "a signed integer", specifiers(s, i + 2));
      when "u" do return _cons(uint, "an unsigned integer", specifiers(s, i + 2));
      when "n" do return _cons(numeric, "a numeric value", specifiers(s, i + 2));
      otherwise do compilerError("Invalid format string: unknown format type");
    }
  } else {
    return specifiers(s, i + 1);
  }
}
```

Extracting Types from Format Strings

Let's give it a quick try:

```
writeln(specifiers("Hello, %s! Your ChapelCon submission is #%i\n") : string);
```

The above prints:

```
_cons(string,"a string",_cons(int(64),"a signed integer",_nil))
```

Validating Argument Types

- The Chapel standard library has a nice `isSubtype` function that we can use to check if an argument matches the expected type.
- Suppose the `.length` of our type specifiers matches the number of arguments to `printf`
- Chapel doesn't currently support empty tuples, so if the lengths match, we know that `specifiers` is non-empty.
- Then, we can validate the types as follows:

```
proc validate(type specifiers: _cons(?t, ?s, ?rest), type argTup, param idx) {
  if !isSubtype(argTup[idx], t) then
    compilerError("Argument " + (idx + 1) : string + " should be " + s + " but got " + argTup[idx]:string, idx+2);

  if idx + 1 < argTup.size then
    validate(rest, argTup, idx + 1);
}
```

- The `idx+2` argument to `compilerError` avoids printing the recursive `validate` calls in the error message.

The `fprintln` overloads

- I named it `fprintln` for "formatted print line".
- To support the empty-specifier case (Chapel varargs don't allow zero arguments):

```
proc fprintln(param format: string) where specifiers(format).length == 0 {
  writeln(format);
}
```

- If we do have type specifiers, to ensure our earlier assumption of `size` matching:

```
proc fprintln(param format: string, args...)
  where specifiers(format).length != args.size {
  compilerError("'fprintln' with this format string expects " +
                specifiers(format).length : string +
                " argument(s) but got " + args.size : string);
}
```

The `fprintln` overloads

- All that's left is the main `fprintln` implementation:

```
proc fprintln(param format: string, args...) {
  validate(specifiers(format), args.type, 0);

  writef(format + "\n", (...args));
}
```

Using `fprintln`

```
fprintln("Hello, world!");         // fine, prints "Hello, world!"
fprintln("The answer is %i", 42); // fine, prints "The answer is 42"

// compiler error: Argument 3 should be a string but got int(64)
fprintln("The answer is %i %i %s", 1, 2, 3);
```

More work could be done to support more format specifiers, escapes, etc., but the basic idea is there.

Beyond Lists

Beyond Lists

- I made grand claims earlier
  - "Write functional-ish program at the type level!"
- So far, we've just used lists and some recursion.
- Is that all there is?

Algebraic Data Types

- The kinds of data types that Haskell supports are called *algebraic data types*.
- At a fundamental level, they can be built up from two operations: *Cartesian product* and *disjoint union*.
- There are other concepts to build recursive data types, but we won't need them in Chapel.
  - To prove to you I know what I'm talking about, some jargon: *initial algebras*, *the fixedpoint functor*, *catamorphisms*...
  - Check out *Bananas, Lenses, Envelopes and Barbed Wire* by Meijer et al. for more.
- **Claim**: Chapel supports disjoint union and Cartesian product, so we can build any data type that Haskell can.

Algebraic Data Types

- The kinds of data types that Haskell supports are called *algebraic data types*.
- At a fundamental level, they can be built up from two operations: *Cartesian product* and *disjoint union*.
- There are other concepts to build recursive data types, but we won't need them in Chapel.
  - To prove to you I know what I'm talking about, some jargon: *initial algebras*, *the fixedpoint functor*, *catamorphisms*...
  - Check out *Bananas, Lenses, Envelopes and Barbed Wire* by Meijer et al. for more.
- **Claim**: Haskell supports disjoint union and Cartesian product, so we can build any data type that Haskell can.

A General Recipe

To translate a Haskell data type definition to Chapel:

- For each constructor, define a `record` with that constructor's name
- The fields of that record are `type` fields for each argument of the constructor
  - If the argument is a value (like `Int`), you can make it a `param` field instead
- A visual example, again:

```
record C1 { type arg1; /* ... */ type argi; }
// ...
record Cn { type arg1; /* ... */ type argj; }
```

```
data T = C1 arg1 ... argi
       | ...
       | Cn arg1 ... argj
```

Inserting and Looking Up in a BST

```
proc insert(type t: Empty, param x: int) type do return Node(x, Empty, Empty);
proc insert(type t: Node(?v, ?left, ?right), param x: int) type do
    select true {
      when x < v do return Node(v, insert(left, x), right);
      otherwise do return Node(v, left, insert(right, x));
    }

type test = insert(insert(insert(Empty, 2), 1), 3);



proc lookup(type t: Empty, param x: int) param do return false;
proc lookup(type t: Node(?v, ?left, ?right), param x: int) param do
    select true {
      when x == v do return true;
      when x < v do return lookup(left, x);
      otherwise do return lookup(right, x);
    }
```

```
insert :: Int -> BSTree -> BSTree
insert x Empty = Node x Empty Empty
insert x (Node v left right)

    | x < v     = Node v (insert x left) right
    | otherwise = Node v left (insert x right)


test = insert 3 (insert 1 (insert 2 Empty))


lookup :: Int -> BSTree -> Bool
lookup x Empty = False
lookup x (Node v left right)

    | x == v    = True
    | x < v     = lookup x left
    | otherwise = lookup x right
```

It really works!

```
writeln(test : string);
// prints Node(2,Node(1,Empty,Empty),Node(3,Empty,Empty))

writeln(lookup(test, 1));
// prints true for this one, but false for '4'
```

A Key-Value Map

```
record Empty {}
record Node { param key: int; param value; type left; type right; }

proc insert(type t: Empty, param k: int, param v) type do return Node(k, v, Empty, Empty);
proc insert(type t: Node(?k, ?v, ?left, ?right), param nk: int, param nv) type do
    select true {
      when nk < k do return Node(k, v, insert(left, nk, nv), right);
      otherwise do return Node(k, v, left, insert(right, nk, nv));
    }

proc lookup(type t: Empty, param k: int) param do return "not found";
proc lookup(type t: Node(?k, ?v, ?left, ?right), param x: int) param do
    select true {
      when x == k do return v;
      when x < k do return lookup(left, x);
      otherwise do return lookup(right, x);
    }

type test = insert(insert(insert(Empty, 2, "two"), 1, "one"), 3, "three");
writeln(lookup(test, 1)); // prints "one"
writeln(lookup(test, 3)); // prints "three"
writeln(lookup(test, 4)); // prints "not found"
```

Conclusion

- Chapel's type-level programming is surprisingly powerful.
- We can write compile-time programs that are very similar to Haskell programs.
- This allows us to write highly parameterized code without paying runtime overhead.
- This also allows us to devise powerful compile-time checks and constraints on our code.
- This approach allows for general-purpose programming, which can be applied to `your use-case`

Extra Slides

Linear Multi-Step Method Approximator

Euler's Method

A first-order differential equation can be written in the following form:

$$y' = f(t, y)$$

In other words, the derivative of of y depends on t and y itself. There is no solution to this equation in general; we have to approximate.

If we know an initial point (t_0, y_0), we can approximate other points. To get the point at $t_1 = t_0 + h$, we can use the formula:

$$\begin{aligned} y'(t_0) &= f(t_0, y_0) \\ y(t_0 + h) &\approx y_0 + h \times y'(t_0) \\ &\approx y_0 + h \times f(t_0, y_0) \end{aligned}$$

We can name the first approximated y-value y_1, and set it:

$$y_1 = y_0 + h \times f(t_0, y_0)$$

Euler's Method

On the previous slide, we got a new point (t_1, y_1). We can repeat the process to get y_2:

$$
\begin{gathered}
y_2 = y_1 + h \times f(t_1, y_1) \\
y_3 = y_2 + h \times f(t_2, y_2) \\
y_4 = y_3 + h \times f(t_3, y_3) \\
\cdots \\
y_{n+1} = y_n + h \times f(t_n, y_n)
\end{gathered}
$$

Euler's Method in Chapel

This can be captured in a simple Chapel procedure:

```
proc runEulerMethod(step: real, count: int, t0: real, y0: real) {
    var y = y0;
    var t = t0;
    for i in 1..count {
        y += step*f(t,y);
        t += step;
    }
    return y;
}
```

Other Methods

- In Euler's method, we look at the slope of a function at a particular point, and use it to extrapolate the next point.
- Once we've computed a few points, we have more information we can incorporate.
  - When computing y_2, we can use both y_0 and y_1.
  - To get a good approximation, we have to weight the points differently.

$$y_{n+2} = y_{n+1} + h\left(\frac{3}{2}f(t_{n+1}, y_{n+1}) - \frac{1}{2}f(t_n, y_n)\right)$$

  - More points means better accuracy, but more computation.
- There are other methods that use more points and different weights.
  - Another method is as follows:

$$y_{n+3} = y_{n+2} + h\left(\frac{23}{12}f(t_{n+2}, y_{n+2}) - \frac{16}{12}f(t_{n+1}, y_{n+1}) + \frac{5}{12}f(t_n, y_n)\right)$$

Generalizing Multi-Step Methods

Explicit Adams-Bashforth methods in general can be encoded as the coefficients used to weight the previous points.

Method	Equation	Coefficient List
Euler's method	$y_{n+1} = y_n + h \times f(t_n, y_n)$	1
Two-step A.B.	$y_{n+2} = y_{n+1} + h\left(\frac{3}{2} f(t_{n+1}, y_{n+1}) - \frac{1}{2} f(t_n, y_n)\right)$	$\frac{3}{2}, -\frac{1}{2}$

Generalizing Multi-Step Methods

Explicit Adams-Bashforth methods in general can be encoded as the coefficients used to weight the previous points.

Method	Equation	Chapel Type Expression
Euler's method	$y_{n+1} = y_n + h \times f(t_n, y_n)$	`Cons(1,Nil)`
Two-step A.B.	$y_{n+2} = y_{n+1} + h\left(\frac{3}{2}f(t_{n+1}, y_{n+1}) - \frac{1}{2}f(t_n, y_n)\right)$	`Cons(3/2,Cons(-1/2, Nil))`

Supporting Functions for Coefficient Lists

```
proc length(type x: Cons(?w, ?t)) param do return 1 + length(t);
proc length(type x: Nil)          param do return 0;

proc coeff(param x: int, type lst: Cons(?w, ?t)) param where x == 0 do return w;
proc coeff(param x: int, type lst: Cons(?w, ?t)) param where x  > 0 do return coeff(x-1, t);
```

A General Solver

```
proc runMethod(type method, h: real, count: int, start: real,
               in ys: real ... length(method)): real {
```

- `type method` accepts a type-level list of coefficients.
- `h` encodes the step size.
- `start` is t_0, the initial time.
- `count` is the number of steps to take.
- `in ys` makes the function accept as many `real` values (for $y_0, y_1, \ldots$) as there are weights

A General Solver

```
param coeffCount = length(method);
// Repeat the methods as many times as requested
for i in 1..count {
    // We're computing by adding h*b_j*f(...) to y_n.
    // Set total to y_n.
    var total = ys(coeffCount - 1);
    // 'for param' loops are unrolled at compile-time -- this is just
    // like writing out each iteration by hand.
    for param j in 1..coeffCount do
        // For each coefficient b_j given by coeff(j, method),
        // increment the total by h*bj*f(...)
        total += step * coeff(j, method) *
            f(start + step*(i-1+coeffCount-j), ys(coeffCount-j));
    // Shift each y_i over by one, and set y_{n+s} to the
    // newly computed total.
    for param j in 0..< coeffCount - 1 do
        ys(j) = ys(j+1);
    ys(coeffCount - 1) = total;
}
// return final y_{n+s}
return ys(coeffCount - 1);
```

Using the General Solver

```
type euler = cons(1.0, empty);
type adamsBashforth = cons(3.0/2.0, cons(-0.5, empty));
type someThirdMethod = cons(23.0/12.0, cons(-16.0/12.0, cons(5.0/12.0, empty)));
```

Take a simple differential equation $y' = y$. For this, define `f` as follows:

```
proc f(t: real, y: real) do return y;
```

Now, we can run Euler's method like so:

```
writeln(runMethod(euler, step=0.5, count=4, start=0, 1)); // 5.0625
```

To run the 2-step Adams-Bashforth method, we need two initial values:

```
var y0 = 1.0;
var y1 = runMethod(euler, step=0.5, count=1, start=0, 1);
writeln(runMethod(adamsBashforth, step=0.5, count=3, start=0.5, y0, y1)); // 6.02344
```

The General Solver

We can now construct solvers for any explicit Adams-Bashforth method, without writing any new code.

Cartesian Product

For any two types, the *Cartesian product* of these two types defines all pairs of values from these types.

- This is like a two-element tuple *at the value level* in Chapel.
- We write this as $A \times B$ for two types A and B.
- In (type-level) Chapel and Haskell:

```
record Pair {
    type fst;
    type snd;
}

type myPair = Pair(myVal1, myVal2);
```

```
data Pair = MkPair
    { fst :: A
    , snd :: B
    }

myPair = MkPair myVal1 myVal2
```

Disjoint Union

For any two types, the *disjoint union* of these two types defines values that are either from one type or the other.

- This is *almost* like a `union` in Chapel or C...
- But there's extra information to tell us which of the two types the value is from.
- We write this as $A + B$ for two types A and B.
- In Chapel and Haskell:

```
record InL { type value; }
record InR { type value; }

type myFirstCase = InL(myVal1);
type mySecondCase = InR(myVal2);
```

```
data Sum
  = InL A
  | InR B

myFirstCase = InL myVal1
mySecondCase = InR myVal2
```

Algebraic Data Types

- We can build up more complex types by combining these two operations.
  - Need a triple of types A, B, and C? Use $A \times (B \times C)$.
  - Similarly, "any one of three types" can be expressed as $A + (B + C)$.
  - A `Option<T>` type (in Rust, or `optional<T>` in C++) is $T + \text{Unit}$.
    - `Unit` is a type with a single value (there's only one `None` / `std::nullopt`).
- Notice that in Chapel, we moved up one level

Thing	Chapel	Haskell
`Nil`	type	value
`Cons`	type constructor	value constructor
List	**???**	type

Algebraic Data Types

- Since Chapel has no notion of a type-of-types, we can't enforce that our values are *only* `InL` or `InR` (in the case of `Sum`).
- This is why, in Chapel versions, type annotations like `A` and `B` are missing.

```
record Pair {
    type fst; /* : A */
    type snd; /* : B */
}
```

```
data Pair = MkPair
    { fst :: A
    , snd :: B
    }
```

- So, we can't enforce that the user doesn't pass `int` to our `length` function defined on lists.
- We also can't enforce that `InL` is instantiated with the right type.
- So, we lose some safety compared to Haskell...
- ...but we're getting the compiler to do arbitrary computations for us at compile-time.

Worked Example: Binary Search Tree

In Haskell, binary search trees can be defined as follows:

```
data BSTree = Empty
            | Node Int BSTree BSTree

balancedOneTwoThree = Node 2 (Node 1 Empty Empty) (Node 3 Empty Empty)
```

Written using Algebraic Data Types, this is:

$$\text{BSTree} = \text{Unit} + (\text{Int} \times \text{BSTree} \times \text{BSTree})$$

In Haskell (using sums and products):

```
type BSTree' = Unit `Sum` (Int `Pair` (BSTree' `Pair` BSTree'))

balancedOneTwoThree' = InR (2 `MkPair` (InR (1 `MkPair` (InL MkUnit `MkPair` InL Unit)) `MkPair`
                                        InR (3 `MkPair` (InL MkUnit `MkPair` InL Unit))))
```

Worked Example: Binary Search Tree

- Recalling the Haskell version:

```
type BSTree' = Unit `Sum` (Int `Pair` (BSTree' `Pair` BSTree'))

balancedOneTwoThree' = InR (2 `MkPair` (InR (1 `MkPair` (InL MkUnit `MkPair` InL Unit)) `MkPair`
                                        InR (3 `MkPair` (InL MkUnit `MkPair` InL Unit))))
```

- We can't define `BSTree'` in Chapel (no type-of-types), but we can define `balancedOneTwoThree'`:

```
type balancedOneTwoThree =
  InR(Pair(2, Pair(InR(Pair(1, Pair(InL(), InL()))),
                   InR(Pair(3, Pair(InL(), InL())))))));
```

- ✅ We can use algebraic data types to build arbitrarily complex data structures ⬛.

Returning to Pragmatism

- We could've defined our list type in terms of `InL`, `InR`, and `Pair`.
- However, it was cleaner to make it look more like the non-ADT Haskell version.
- Recall that it looked like this:

```
record Nil {}
record Cons { param head: int; type tail; }

type myList = Cons(1, Cons(2, Cons(3, Nil)));
```

```
data ListOfInts = Nil
                | Cons Int ListOfInts

myList = Cons 1 (Cons 2 (Cons 3 Nil))
```

- We can do the same thing for our binary search tree:

```
record Empty {}
record Node { param value: int; type left; type right; }

type balancedOneTwoThree = Node(2, Node(1, Empty, Empty),
                                   Node(3, Empty, Empty));
```

```
data BSTree = Empty
            | Node Int BSTree BSTree

balancedOneTwoThree = Node 2 (Node 1 Empty Empty)
                             (Node 3 Empty Empty)
```