CS 252r: Advanced Topics in Programming Languages

Eric K. Zhang
ekzhang@college.harvard.edu

Fall 2021

Abstract

These are notes for Harvard's *CS 252r*, a graduate seminar class on the design and implementation of programming language systems, as taught by Nada Amin[1] in Fall 2021. Topics include reflection, verification, relational/logic programming, synthesis, compilation, probabilistic programming, differentiable programming, and others.

Course description: A hands-on exploration of programming language systems for various purposes. Each student will design and implement their own system (for example, a virtual machine and/or programming language and/or framework) in the language of their choice for the purpose of their choice.

Contents

[1] With teaching fellow: Anastasiya Kravchuk-Kirilyuk

1 September 2nd, 2021

Today is the first lecture of the course. There are about 11 students in the seminar room.

1.1 Introduction

This class discusses *PL systems*, i.e., any system that either uses programming language techniques or is used for domain-specific language purposes. There are two requirements in this class; first, two mini-projects at the beginning (microKanren and an SMT-based Sudoku solver); and second, a final project. The topic is extending or combining papers in the PL design literature.

First, we ask the question: **what is a programming language?** What differentiates a language from a library, especially in edge cases like PPLs, which tend to be implemented or embedded in a host language? They key axis is that programming languages allow you to do **many things**, while libraries tend to let you do **one or a few things** well. Furthermore, programming languages have specifications and implementations, while libraries tend to only have one implementation. Oftentimes there are reasons for which we want to write a new programming language, focused on tradeoffs between what is easy versus difficult too represent.

Examples of PL systems are Dafny [Lei10a], LMS [RO10], Halide [RKBA+13], miniKanren [Byr09], and Ur/Web [Chl16]. The research cycle is design, implement, and evaluate. From [AS96], we can think of languages as being defined by their primitive components and methods of creating new abstractions. As a simple example, the Pan language has the key idea of representing images as functions (shallow embedding in JavaScript), which allows us to abstract over images and create higher-order combinators.

1.2 Staging and DSLs

One of the key techniques in programming languages design is *staging*. Consider the example of a regular expression library, which may act as an interpreter to match certain string patterns. To turn this interpreter into a compiler, we can *stage* the computation by performing optimized code generation on a known regular expression, even before knowing the input text to search over. This may result in performance benefits if we use the regular expression many times. Nada presents an example of writing a staged regular expression compiler in Scala using LMS.

General-purpose languages are meant to do many things well, but they have a bottleneck in that everything passes through a single compiler, which may not have domain knowledge that is useful for expressivity or optimization. In contrast, for *domain-specific languages*, there are a couple of evaluation metrics:

- Generalizes expression of specific problems, but does not over-generalize.
- Separates declarative and implementation aspects, like optimization.
- Conceptual integrity in few but powerful, composable concepts.
- Exploits domain knowledge for performance.

One example of where domain-specific languages are useful is in formal verification. Consider Frama-C [CKK+12], which is a popular low-level formal verification system for C that checks memory safety and correctness using logical annotations added to comments in C code.

Although logical annotations are powerful, they can be very tedious to write by hand, and so systems like LMS-Verify [AR17] can build on staged code generation to add formal verification

properties to those systems. For example, we can modify a staged compiler for parser combinators to automatically generate efficient, secure C code for an HTTP parser in Scala.

1.3 Dependent Object Types

Now, we shift focus to discussing *dependent object types (DOT)*, which are a type-theoretic foundation for Scala 3. This covers the majority of the Scala type system (and "simplified" it quite a bit), but it does not cover higher-kindred types, which is still an open problem. Knowing this theory leads to fun results like the the unsoundness of Java and Scala [AT16].

In dependent object types systems, we have a few primitives and means of combination. In the following bullet points, S is contravariant, and U is covariant:

- The top and bottom types $\top, \bot$, function types $S \to U$, and type bounds $S \leq T \leq U$.
- Undiscriminated union types and intersection types: $T \cup U$ and $T \cap U$.
- Means of abstraction, by specifying associated types in traits: $\{z \Rightarrow T\}$, and path-dependent types: $x.L$, which are hidden within records.

Dependent object types are implemented in Scala using a solver based on Coq. They allow you to define traits with associated types **depending on the value** of the object, which are a key means of abstraction that gives you additional expressive power, such as below.[2]

```
1  trait X {
2    type Image[T] // this is a higher-kinded dependent type
3    def combine[T](a: Image[T], b: Image[T]): Image[T]
4  }
5
6  trait Y: X {
7    type Image[T] = Pixel => T
8    // ...
9  }
```

This shows how DOT as a system falls into the structural pattern of primitives, means of combination, and means of abstraction.

[2]Disclaimer: I am not a Scala programmer and probably made mistakes while scribing this.

2 September 7th, 2021

Today we will have a guest lecture from Will Byrd about *miniKanren*, a Scheme-embedded relational language for logic programming.

2.1 Relational Programming

Relational programming is a *paradigm*, similar to but lesser-known than other paradigms like object-oriented programming, functional programming, stack-based programming, logic programming, and deep learning. Just like how functional programming focuses on functions as the unit of abstraction, relational programming focuses on representing mathematical concepts as *relations*.

Note. With some modern programming paradigms, like deep learning, there is a break between the paradigm of the host language (Python, procedural) and the actual concept being represented. Oftentimes, we hear the phrase "library design is language design." Will does not agree completely with this statement, as part of good language design is using the *weakest language possible*, like having regular expressions to represent certain string languages.

After a brief introduction to Scheme, Will gives us a demonstration of Barliman, a prototype program synthesis tool based on miniKanren that can generate Scheme programs from tests of their expected behavior. It uses constraint solving and search algorithms to fill in gaps in the program based on operational semantics.

```
; Input - Definitions
(define ,A
  (lambda (l s)
    ,C))

; Input - Test 1
(== (append '() '()) '())

; Input - Test 2
(== (append '(,g1) '(,g2)) `(,g1 ,g2))

; Input - Test 3
(== (append '(,g3 ,g4) '(,g5 ,g6)) `(,g3 ,g4 ,g5 ,g6))

; Output - Best Guess
(define append
  (lambda (l s)
    (if (null? l) s (cons (car l) (append (cdr l) s)))))
```

You can provide other constraints to the definitions, and miniKanren will be able to seamlessly synthesize programs that satisfy those constraints. However, it is unclear whether adding additional structure to the `(append)` program template actually improves performance, since many of the "optimizations" in to miniKanren's search system are actually heuristics.

Generally, the goal of such relational programming systems is to allow for meta-variables within programs, and they are focused on allowing people to solve for more generic and complex programs. We will implement a custom constraint solver later in this course as homework, based on a simple and minimal variant called microKanren [HF13].

2.2 The miniKanren System [Byr09]

Recall that miniKanren is an *embedded* domain specific language within Scheme. In order to run miniKanren code, you need to write some special Scheme syntax in order to execute the solver and produce results. The system offers a macro `==` that returns some symbolic representation of the constraint, which is used by the miniKanren interpreter `run`.

```
; Input 1
(run 1 (q)
  (== (list 5 q)
      (list 5 7)))
; Output 1
(7)

; Input 2
(run 1 (x y)
  (== (list 5 x)
      (list 5 y)))
; Output 2
((_.0, _.0))

; Input 3
(run 1 (x y)
  (== (list 5 x)
      (list y 6)))
; Output 3
((6 5))

; Input 4
(run 1 (x y w v)
  (== (list w x)
      (list y v)))
; Output 4
((_.0 _.1 _.1 _.0))
```

Notice that the outputs with wildcard `_.0` syntax do not have explicit equality constraints, due to *unification*, an optimization that is automatically applied by the miniKanren interpreter.

```
; Input 5
(run 1 (x y)
  (=/= x y))
; Output 5
(((_.0 _.1) (=/= _.0 _.1)))
```

Here, we used new syntax in the `=/=` *disequality* operator, which is fundamentally different from our previous `==` equality constraints. The program appropriately generated an input with an extra constraint between the variables `_.0` and `_.1`. We can attempt to unify the variables with an extra condition, but then miniKanren will detect that there are no satisfiable substitutions and fail.

```
; Input 6
(run 1 (x y)
  (=/= x y)
  (== x y))
; Output 6
()
```

What happens if we add a disjunctive logical clause? We can do this using the `conde` core operator. After adding disjunction, we may begin to have multiple satisfying assignments, so the `run*` operator allows us to provably obtain all successful assignments.

```
; Input 7
(run 1 (x)
  (conde
    ((== x 5))
    ((== x 6))))
; Output 7
(5)

; Input 8
(run* (x)
  (conde
    ((== x 5))
    ((== x 6))))
; Output 8
(5 6)
```

There is one more operator called `fresh`, which allows you to define new auxiliary variables within a miniKanren program. Also, miniKanren (like many logic programming systems) has the nice property that reordering the arguments in logical clauses like `conde` and `==` does not change the operational semantics of the program.[3]

```
; Input 9
(run* (x y)
  (conde
    ((== y 7) (== 5 x))
    ((== 6 x) (== x y))))
; Output 9
((5 7) (6 6))
```

Finally, the last detail about core operators is `conde`, which actually functions as a disjunction of conjunctions for its inputs. In the example above, we passed in a DNF expression, and the result returned satisfying assignments from either of the two clauses.

Note. In the following example, it is very important that the recursive `appendo` relation is placed at the end of the constraint list. Otherwise, the program will loop indefinitely rather than failing

[3]This is assuming termination. Sometimes reordering conjunction will make a failure program non-terminating.

while attempting to search for more satisfying results, rather than narrowing down its search. This is similar to how left recursion can cause recursive descent parsers to produce an infinite loop.

```
; Input 10
(define appendo
  (lambda (l s out)
    (conde
      ((== '() l) (== s out))
      ((fresh (hd tl res)
         (== `(,hd . ,tl) l)
         (== `(,hd . ,res) out)
         (appendo tl s res))))))

; Input 11
(run* (x y)
  (appendo x y '(a b c d e)))

; Output 11
((() (a b c d e))
  ((a) (b c d e))
  ((a b) (c d e))
  ((a b c) (d e))
  ((a b c d) (e))
  ((a b c d e) ()))
```

Going back to program synthesis, one fascinating application of logic programming is, what if we were to write a Scheme interpreter within miniKanren? The result, which we call `evalo`, can be used to synthesize terms of Scheme programs based on their behavior on input/output pairs. This is exactly what is happening in Barliman, on a low level.

3 September 9th, 2021

Today, we have a second lecture from Will Byrd about relational programming with miniKanren.

3.1 More on miniKanren

To recap from last lecture, we showed how we can use relational programming to define predicates like `appendo`, and the result is that we can synthesize the inputs to produce a desired output of the function, effectively solving an inverse problem.

```
; Input 1
(run* (q)
  (appendo '(a b c) `(,q e) '(a b c d e)))

; Output 1
(d)
```

What if we wanted to actually synthesize the `appendo` predicate? It turns out that we cannot do this with normal miniKanren, because `appendo` is a *goal* within the relational system rather than something to be searched over. The `==` operator only implements *first-order unification* of terms like lists, symbols, and numbers, and it cannot unify relations like `appendo` themselves.[4]

```
; Input 2
(run* (q)
  (list q '(a b c) '(d e) '(a b c d e)))

; Output 2
Exception...
```

To search over spaces like this, Will shows us his current work on building an interpreter for μKanren that is written in miniKanren. This comes in the form of a predicate called `eval-programo`, which is able to synthesize μKanren programs from within a miniKanren expression using logic variables.

Note. Although this is a very powerful idea, it has the limitation that every level of meta-relational programming adds significant interpretive overhead to the system. This requires optimization to make problems tractable.

Right now, the system can only handle one level of nesting μKanren within faster-miniKanren, but if one were to write a self-hosted interpreter for miniKanren, this could offer unlimited levels of nesting (albeit with increasing overhead). Maybe, using staged compilation, it could be possible to "collapse infinite towers" of miniKanren interpreters.

Finally, Will presents an example of a fun application of miniKanren to finding *quines*. One of the most famous exercises in Lisp is to find an expression e that evaluates to itself. This is a tricky mathemtical exercise to do manually, but miniKanren with the Scheme `evalo` interpreter is able to find this easily. Note that in the following query, it does not find trivial quines like `#t`, `#f`, and `42`, as this interpreter only supports a partial subset of Scheme.

[4] There are some languages that offer higher-order unification of relations, such as λ-Prolog [NM88].

```
; Input 3
(run* (e)
  (evalo e e))

; Output 3
((((lambda (_.0) (list _.0 (list 'quote _.0)))
    '(lambda (_.0) (list _.0 (list 'quote _.0))))
  (=/= ((_.0 closure)) ((_.0 list)) ((_.0 quote)))
  (sym _.0)))
```

We can even find *twines*, which are expressions that first become themselves again after two evaluation passes.

```
; Input 4
(run* (p q)
  (=/= p q)
  (evalo p q)
  (evalo q p))

; Output 4
...
```

The final example is extending this Scheme interpreter in miniKanren with quasi-quoting support. Using staged evaluation, we can write a Scheme interpreter for quasi-quotes within Scheme, which is then run in miniKanren's interpreter for Scheme. This allows us to find pretty quines like:

```
((lambda (x) `((lambda (x) ,x) ',x)) '`((lambda (x) ,x) ',x))
```

3.2 Implementation of microKanren [HF13]

We discuss some details about the microKanren implementation in `microKanren.scm`:

- **Why are variables represented by vectors?** This is just an arbitrary choice; we can use any syntax not directly supported by our language (cons pairs, atoms, numbers) to represent variables. Here, we store a counter as the first item of the vector, but we could also just heap-allocate new vectors and compare them for pointer equality with `eq?`.
- **Why does the `walk` function recurse?** This is because we use *triangular substitution* in this interpreter, which allows variables to refer to other variables. This makes lookup slower, as it might require multiple steps, but it makes building the state easier. There is another representation called *idempotent substitution*, where every lookup only requires one access into the substitution list `s`.
- **How do goals work?** A goal is represented by a function that takes a pair of substitution and counter values, abbreviated `s/c`. On success, it returns a stream of substitution and counter pairs, which are suitable to satisfy the goal. Otherwise, it returns an empty stream `mzero` on failure.

- **What is the difference between fresh and call/fresh?** The microKanren variant uses `call/fresh` because it does not add extra syntactic sugar on top of Scheme with macros, so `(fresh (q) ...)` is represented by `(call/fresh (lambda (q) ...))`. However, we might prefer `fresh` for a more polished implementation, since it allows us to write pure miniKanren code without leaking implementation details of the host language like `(lambda)` forms.

The first assignment of this course, to be interpreted liberally, is to implement your own μKanren within some choice of host language other than Scheme.

4 September 14th, 2021

Today we demonstrate how to use SMT solvers, using Z3 as our solver of choice.

4.1 Introduction to Z3 [DMB08]

We introduce the features of SMT solvers through SMT-LIB 2, which is a common interface for solvers like Z3, Boolector, and CVC4. Let's first see how we can use SMT solvers to solve SAT, i.e., propositional logic problems.

```
(declare-const a Bool)
(declare-const b Bool)

(assert (and a b))
(check-sat) ;; sat
(get-model)

(push)
(assert (not a))
(check-sat) ;; unsat
(pop)

(check-sat) ;; sat
```

This program can be run with the command `z3 live.smt2`. Notice how the `check-sat` function allows us to solve problems, while the `get-model` function is useful for actually finding a satisfiable assignment. Also, we can generate new stack frames with `push`, which is useful for algorithms like depth-first search and symbolic execution.

Of course, SMT solvers are not just capable of solving SAT. The acronym SMT stands for *satisfiability modulo theories*, meaning that SMT solvers provide many logical theories that allow us to solve constraints on different types of values. These have decidability rules that determine how they are lowered to propositional logic. Common theories in Z3 include arithmetic, arrays, uninterpreted functions (equality), bit-vectors, and strings.

Note. The way that these theories are actually represented by the solver is by translating some approximation of the constraints to propositional logic, asking the SAT solver if it is satisfiable, then progressively adding finer constraints.

Here's another example, using theories of linear arithmetic and uninterpreted functions.

```
(declare-fun f (Int) Int)
(declare-fun a () Int) ; equivalent to (declare-const a Int)
(declare-const b Int)

(assert (> a 20))
(assert (> b a))
(assert (= (f 10) 1))

(check-sat) ;; sat
(get-model)
```

Notice that the theory of linear arithmetic already provides us with all the representative power of an LP solver, and more, though it might not be as efficient as a tailored optimizer. Another common theory is bit-vectors, which allows us to represent finite sets. We can also define new record data types within SMT-LIB 2.

```
;; {first: T1, second: T2}: Pair T1 T2
(declare-datatypes (T1 T2) ((Pair (mk-pair (first T1) (second T2)))))
(declare-const p1 (Pair Int Int))
(declare-const p2 (Pair Int Int))

(assert (= p1 p2))
(assert (> (second p1) 20))
(check-sat) ;; sat
(get-model)
;; => (
;;   (define-fun p2 () (Pair Int Int)
;;     (mk-pair (0 21))
;;   (define-fun p1 () (Pair Int Int)
;;     (mk-pair (0 21))
;; )

(assert (not (= (first p1) (first p2))))
(check-sat) ;; unsat
```

Note that the `declare-datatypes` syntax allows us to create general algebraic data types, so we can create enums with a similar syntax.

```
;; enum S { A, B, C }
(declare-datatypes () ((S A B C)))
(declare-const x S)
(declare-const y S)
(declare-const z S)
(declare-const w S)

(assert (distinct x y z))
(check-sat) ;; sat

(assert (distinct x y z w))
(check-sat) ;; unsat
```

We can also encode recursive data types like lists, and mutually recursive data types, using this syntax. This allows us to reason about lists, which is very powerful!

```
(declare-datatypes (T) ((List nil (cons (hd T) (tl List)))))
(declare-const l1 (List Int))

(declare-datatypes (T)
  ((Tree (T) (node (children TreeList)))
   (TreeList (T) nil (cons (car Tree) (cdr TreeList)))))
```

The nice thing about SMT-LIB 2 is that it standardizes a black-box interface for a ton of SMT solvers that apply very sophisticated strategies for solving constraints. It offers benchmarks that all of the solvers target, and it lets applications build on top of a common interface.

However, there are limitations to some SMT solving interfaces. For example, one can construct an infinite loop by asking Z3 to prove a statement about natural numbers by induction. To avoid these situations in practice, we can set a timeout. If the SMT solver hits a timeout, it will return `unknown` to the `check-sat` function, rather than `sat` or `unsat`. There is also a syntax where Z3 will allow you to pass *patterns* like `:pattern ((g x))` to a SMT constraint, which may provide hints to the solver and allow it to terminate where it would otherwise not terminate.

There is a certain subset of first-order logic called *effectively propositional* logic, and for this class, the satisfiability problem decidable (albeit NEXPTIME-complete). When the Z3 solver is given a first-order quantified logical expression that lies within this class, it will detect it and use special methods that are guaranteed to terminate.

Finally, we can use the `(declare-sort)` function to declare new general data types that fit in some *universe*. As we add distinct constraints to this data type, the universe will automatically expand to add new values.

4.2 CLP(SMT) miniKanren

By combining SMT solvers with miniKanren, we can add support for theories like general arithmetic to the relational programming that we already support. The code is available here.

```
(run* (q)
  (conde
    ((z/assert `(> ,q 0))
     (z/assert `(< ,q 2)))
    ((z/assert `(= (* ,q 4) 20)))))  ;; (1 5)
```

We can also perform some basic symbolic execution of Scheme. Generally, this system seems to be able to generate many satisfying assignments and is fairly good at program synthesis, but it is still an open question how well miniKanren acts as a driver to SMT solvers.

4.3 Rosette and Symbolic Execution

Rosette [TB14] is a system in Racket for constructing languages that use SMT solvers. It allows us to do things like program synthesis, symbolic execution, and verification with little code, by seamlessly combining SMT solvers with Racket's ability to provide DSL syntax extensions. Similar to other systems, we can use different solvers like Z3 and CVC4 as the backend.

To make things more concrete, we go through a conceptual example of symbolic execution now, presented in Chapter 12 of [KS16]. Essentially, given a program in some language, we can analyze all execution paths through that program and unrolling loops and conditionals. These program paths can be turned into static single assignment (SSA) form, which is translated into a *verification condition* that is passed to the SMT solver interface.

The main issue with this SMT-based verification of imperative languages is *loops* and *recursion*. This is really difficult to verify directly. One way is to over-approximate by unrolling the loop and asserting that it will terminate in at most k iterations, or we could under-approximate by just ignoring the loop. Another more direct approach is to use *loop invariants* and have the SMT solver verify correctness through induction. See Klee [CDE+08] as an real-world example of SMT solvers being used in software engineering to find bugs through symbolic execution.

5 September 16th, 2021

Today, Ana will present a lecture on Dafny [Lei10b], which is a programming language with a built-in verifier for generating provably correct code. If time permits, we will also discuss Coq, a different verifier, for the purpose of comparison.

5.1 Program Verification with Dafny

Let's start by showing an example of a Dafny program.

```
// find key in an array
// if we do not find it, return its index
// if not, return -1

method Find(a: array<int>, key: int) returns (index: int)
  ensures 0 <= index ==> index < a.Length && a[index] == key
  ensures index < 0 ==> forall k :: 0 <= k < a.Length ==> a[k] != key
{
  index := 0;
  while index < a.Length
    invariant 0 <= index <= a.Length
    invariant forall k :: 0 <= k < index ==> a[k] != key
  {
    if a[index] == key { return; }
    index := index + 1;
  }
  index := -1;
}
```

This program compiles because Dafny checks all of the loop invariants and `ensures` clauses through its theorem prover based on Z3, and it is able to deduce that the desired clauses hold. This is a form of **automated program verification** because not every step of the logic needs to be explicitly written down; the user can guide the overall direction while letting Z3 infer in-between steps. Here's another example of a Dafny program, this time with a precondition.

```
method possum(a: array<int>) returns (sum: int)
  requires forall k :: 0 <= k < a.Length ==> a[k] >= 0
  ensures sum >= 0
{
  var i := 0;
  sum := 0;
  while i < a.Length
    invariant sum >= 0
  {
    sum := sum + a[i];
    i := i + 1;
  }
  return; // or: return sum;
}
```

What's perhaps more interesting is if we wanted to define an analogous function for strictly positive sums, rather than just nonnegative sums like the example above. This time, our preconditions would be `forall k :: 0 <= k < a.Length ==> a[k] > 0` and `a.Length > 0`. However, to verify the program, we need to either add some assertions or provide a more complex loop invariant.

Note. As Ana describes, Dafny "knows what it knows," but you as the programmer do not always know what Dafny knows. Therefore, writing and debugging nontrivial Dafny programs typically takes some time. You often need to add extra `assert` statements to help debug. If Dafny is able to verify `assert false;`, then something is wrong!

In addition to standard imperative programs, Dafny is also able to handle code written in a more functional paradigm. Dafny supports algebraic data types and recursion in code. There is also the concept of a *function*, which is different from a method because it can be used within logical clauses. By default, without adding extra conditions or assertions, Dafny just verifies that the function terminates.

```
datatype Tree<T> = Leaf | Node(Tree, T, Tree)

// checks whether a tree contains a value v
function Contains<T>(t: Tree<T>, v: T): bool
{
  match t
  case Leaf => false
  case Node(left, x, right) =>
    x == v || Contains(left, v) || Contains(right, v)
}
```

We can also verify object-oriented code, such as the `SimpleQueue` example from the Dafny paper. The interesting trick in this example is the use of `seq<Data>`, which is a ghost variable that is used to keep track of the verification spec, while not being actually run in the compiled code. It is tied to the value of the underlying array through an always-present `Valid()` invariant.

Note. Dafny lowers programs into an intermediate representation called *Boogie* [BCD+05], which is tailor-made for expressing verification conditions and operates on a superset of C#. This mid-level system then spawns several instances of Z3 to solve constraint problems in parallel.

In addition to the canonical verification pipeline, Dafny now has non-verification runtime backends that lower its code to C#, C++, and Go. However, the essential part of Dafny is still its verification engine, as you often don't even need to run Dafny code to know that it is correct. A clear example is in the case of *lemmas*,[5] which can verify statements at compile time that would otherwise require interpretation (i.e., unit tests) in other programming languages.

5.2 Verifying Streams and STLC

Because of Dafny's semantics based on verification rather than interpretation, you can obtain support for infinite lazy data types using the `codatatype` syntax. This allows you to define functions on things like streams that would usually result in infinite recursion in normal programming languages, but as long as the functions are *productive*, they can be verified by the Dafny system.

[5] Lemmas are equivalent to ghost methods in Dafny.

Finally, we have an example of verifying the simply-typed lambda calculus. This is fairly simple in Dafny, and the verifier can actually infer many of the cases for most lemmas without having to write much code at all! However, you can also verify properties of STLC in a system like this Coq example, which scales better to larger systems (less "magic" that might cause verification issues as soon as you add more complexity) but requires more handwritten logic.

6 September 21th, 2021

Today, Eric Bigelow will present a lecture about Penrose, a recent software platform for producing diagrams from mathematical language. This is our first student presentation.

6.1 Penrose [YNK+20]

The stated goal of Penrose is to provide a reusable notation software that allows mathematicians to produce beautiful, styled diagrams with easy-to-express constraints. In the past, there have been tools for satisfying part of the design process, like GraphViz for visualization or LaTeX for mathematical typesetting. Penrose tries to combine all of these steps into a unified platform with the potential to support arbitrary optimization. There are two steps:

- **Specification:** The user writes a diagram specification in a certain abstract language, which declares symbolic domain, substance, and style rules.
- **Synthesis:** The Penrose solver uses optimization and constraint satisfaction techniques to find diagram layouts that satisfy the specification.

The specification, as mentioned above, consists of three parts. There is a *domain* language `.dsl` for defining abstract data types and predicates (relations) between them, as well as *notations* that provide syntactic sugar for relations. The *substance* language `.sub` has mathematical assertions in the domain, and the *style* language `.sty` translates algebraic relations into layout.

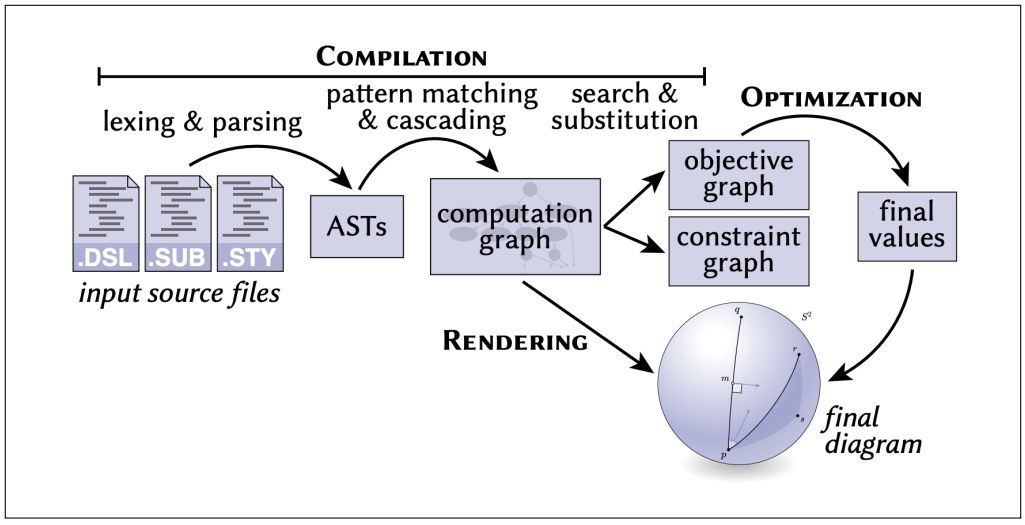

Figure 1: Computation pipeline overview of the Penrose system.

The optimization step uses an *exterior point method*, where they start from an infeasible point and progressively push it towards feasible configurations via progressively stiffer penalty functions. Typically, they sample several initial configurations and only optimize the most promising ones.

Finally, the rest of the magic happens in the rendering step of the pipeline, where location information returned from the optimization step is passed to a React.js / SVG-based rendering engine, which produces all of the pretty diagrammatic output. However, since everything is modular, you could imagine that there would be alternative rendering engines that visualize the output of the initial steps of the Penrose system.

6.2 Thoughts on Penrose

Even though Penrose is still a research system and in development, Eric was able to clone the repository and try making his own diagrams with it. Right now, it's fairly simple to specify the symbolic DSL and substance languages, but it is far harder to produce the style language in a clean way. Furthermore, right now there are some challenges in learning the built-in relations and discovering what functionality is already available to you in the documentation.

Future questions could be: from the graphics side, can we do some kind of inverse graphics to produce the diagrammatic language from visual perception, [ERSLT17]? This would produce some kind of "inverse Penrose" system, which sounds pretty interesting. Are there cognitive analogues to the modular domain, substance, and style components from this paper? Can we do some kind of cognitive science research on the way humans create and interpret drawings?

Max makes the interesting observation that Penrose is very style file-heavy, and the domain is mostly just a definition of the core mathematical structure. However, humans often use visual representations in the first place to think about abstract concepts, so in human cognition, is the core concept really the underlying algebraic structure, or the visual layout?

Finally, it's worth pointing out that the the `.dsl` in Penrose is functionally structured to **reduce the space of possible inputs**, compared to a system like TikZ, where the user has flexibility to basically input anything into their diagram. Similar to sketching for program synthesis (see [SLTB+06]), reducing the search space to structured relational inputs could potentially make inverse problems much more tractable. Turning the problem into a symbolic one also potentially encourages the future use of systems like SMT solvers, so there's a lot of potential future work.

7 September 23rd, 2021

Today, Matt Pereira will present a lecture on dynamic dispatch implemented using prototypal inheritance in a research programming language based on Smalltalk.

7.1 Open, Extensible Object Models [Piu11]

The contribution of this paper is to define a variant of Smalltalk that offers a large number of possible object-oriented design patterns (including inheritance, multiple inheritance, static/dynamic dispatch, and traits), while only relying a minimal set of core message-passing primitives.

The representation of objects in this paper is as a tuple of *state* and *behavior*. Each object contains its data in normal memory addresses, as well as a *vtable pointer* that describes the behavior of the object, including implementations of all methods that can be called on the object. As in Smalltalk, calling a method on an object is equivalent to sending a message to that object via dynamic dispatch on its vtable.

Method names in this system are described using *symbols*. Each symbol is a short string that is interned into a symbol list to avoid extra memory consumption. When we send a symbol to an object, it looks up that symbol in its vtable, and if present, it returns the address of the method that was found. Otherwise, it falls back to the parent vtable and calls lookup there.

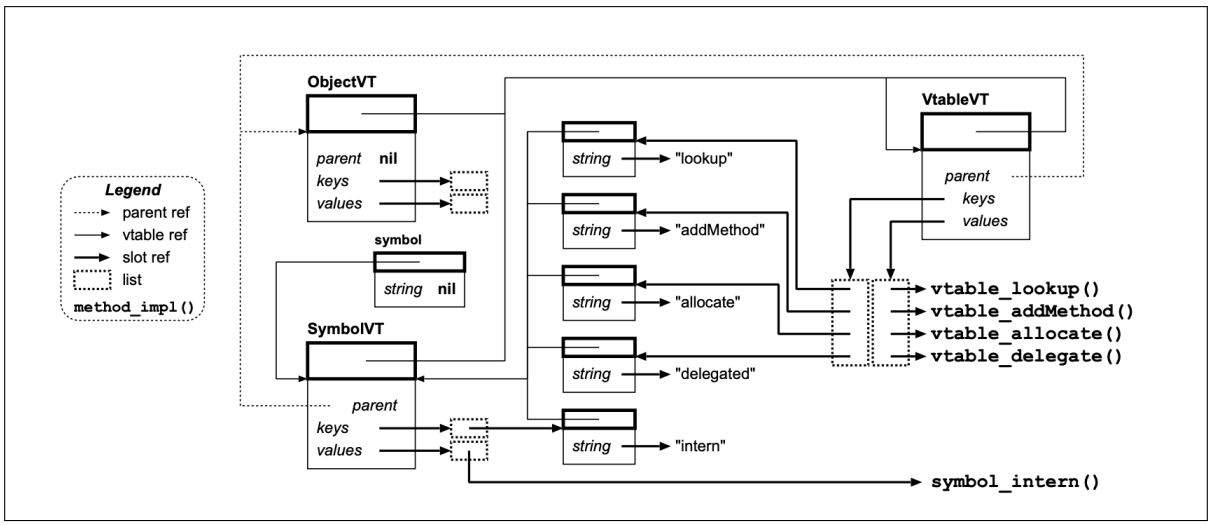

Figure 2: Diagram of vtable and parent object interactions.

Similar to languages like Ruby, there is an interesting interaction where each object has a vtable pointer, but the vtable itself is an object (with parent `ObjectVT`) that has a vtable given by the *vtable's vtable*, denoted `VtableVT`. The core of the black-box engine in this language consists of two functions, `send` and `bind`, which define the behavior of the message-passing framework that all other built-in methods are based on.

```
function send(object, messageName, args...) =
  let method = bind(object, messageName)
  return method(object, args...)
```

To avoid infinite recursive dispatch, we make sure to add a base case for the Vtable's *lookup* function during bootstrapping to make sure that it bottoms out with an eventual method call.

```
function bind(object, messageName) =
  let vt = object[-1]
  let method =
    if messageName = lookup
        and object = VtableVT
      vtable_lookup(vt, lookup)
    else
      send(vt, lookup, messageName)
  return method
```

With these base primitives, we can extend vtables with new delegated vtables (similar to inheritance) by setting the parent and examining this parent within the `vtable_lookup` function. We can also generalize the system to support other models like multiple inheritance, similar to Ruby, and prototypal getter/setter dispatch, similar to JavaScript.

7.2 Performance and Extensions

One section of the paper that is particularly interesting is where they try to support *closures*, which can be used to support mixed-mode execution and pass around behavior to within a function. This shows how functions can be supported as close to first-class values in our language.

Another application of such a system would be as a compilation target of other languages, since its flexibility could be useful for implementing different forms of dispatch, especially when combined with a static type system that prevents some classes of errors.

8 September 28th, 2021

Today, Liam McInroy will present a lecture about Gen, a domain-specific language for flexible probabilistic programming with programmable inference embedded in Julia.

8.1 Probabilistic Programming With Gen [CTSLM19]

In probabilistic programming, there are two main problems. First, *modeling* is the problem of specifying what distribution a set of random variables comes from, possibly parameterized by some vector θ. Second, *inference* roughly consists of the inverse problem, where we want to infer the values of parameters based on the observed posterior.

One of the most common inference problems is Bayesian inference. In a graphical model where we know $\Pr(X_1)$ and $\Pr(X_2 \mid X_1)$, the Bayesian inference task would be to find $\Pr(X_1 \mid X_2)$. This conditional probability distribution can be generatively modeled using *Bayes' rule*, but to scale this computation to complex distributions, it can be fairly intractable. The two main algorithms are:

- **Monte Carlo:** These include methods like MCMC, which allow asymptotically exact sampling from probability distributions using Metropolis-Hastings, Gibbs, Hamiltonian Monte Carlo, or other algorithms. However, these methods can be incredibly inefficient.
- **Variational inference:** A faster method of statistical inference that optimizes an approximation of KL divergence (ELBO) between the true posterior, and a member of a family of probability distributions. This is an inexact inference method.

There are a whole host of programming languages and libraries that support probabilistic programming, generally referred to as PPLs. However, each of these languages tends to have very specialized support for a certain subset of algorithms.

Gen is a system in Julia that hopes to be very flexible and support a wide variety of programmable inference methods. Briefly, there are four main components of Gen's design:

- **Primitive distributions:** The basic, fundamental distributions that are symbolically represented, such as the Bernoulli, Normal, and Gamma distributions. These are deterministic and are used as building blocks in larger programs.
- **Generative functions:** A general interface for specifying probabilistic models with parameters and a resulting distribution, with support for higher-level builtin functions for sampling, taking gradients, resampling, and other common operations.
- **Traces:** A model of a probability distribution that combines the values of several generative functions, which explicitly depend on each other in a graphical format.
- **Combination:** Distributions can depend on each other in push-forward results (deterministic mathematical operations), composition, and four combinators. `map` generates an i.i.d. array of observations, `unfold` applies a single function functorally on sequence data (e.g., Markov chains), `recurse` produces a tree of random variables then combines/reduces them, and `switch` invokes one of a set of generative functions on a given input.

A program written in Gen essentially has two components. First, users specify probability models (parametric distributions) in terms of the *generative function interface (GFI)*, which is declarative and based on Julia's macro system, using `@gen`. Then, they use probabilistic methods on the generative function interface that efficiently automate common probability tasks.

Definition 8.1 (Generative function)**.** A *generative function* $\mathcal{P}$ is a tuple (X, Y, p, f) where:

- X is the input type.
- Y is the output type.
- $p : X \times S \to [0, 1]$ is a family of probability density functions on choice maps $s \in S$ based on an argument $x \in X$, which can be viewed as parameters. Here, s contains a hierarchical, indexed trace of each random choice made by the model.
- $f : \text{supp}(X \times S) \to Y$ is a deterministic function that describes the model's output.

Note that this formal model completely decouples the stochasticity p from the function logic f, which makes it possible to support a wide range of different probabilistic algorithms.

8.2 Probabilistic Combinators in Gen

For optimization reasons, Gen supports three different languages. The first is a Turing complete dynamic modeling language represented by Julia functions annotated with `@gen`, which supports a full range of Julia's operations. There is also a static modeling language, `@gen (static)`, which is amenable to static analysis and optimization (computes minimal subgraph that requires sampling), as well as combinators that provide and optimize common control flow operations.

These combinators are the trick that make inference efficient, since they dramatically simplify the trace graph. Operating on models built upon primitives, functions, and combinators, the GFI operations that Gen allows programmers to use are:

- `simulate`, which executes the function and returns a trace;
- `generate`, which returns a trace that is consistent with the constraints in a choice map;
- `update`, which updates a trace by changing some arguments or random choices;
- `regenerate`, which updates a trace by sampling new values using the proposal distribution;
- `project`, which estimates the conditional probability of a trace;
- `propose`, which samples an assignment and estimates the probability of proposing it;
- `choice_gradients`, which computes gradients of the likelihood function.

There are other GFI methods as well, but these are some of the more important ones. If you want to support additional abstractions on top of the Gen language, you need to provide implementations all of these methods. For example, the `src/modeling_library/map/` folder contains implementations of each method for the `Map` combinator, which is essentially its own sublanguage.

You can define other sublanguages as well, such as the GenPyTorch and GenTF plugins that delegates some of these interface methods to deep learning frameworks. Any inference algorithm in Gen, like Monte Carlo or variational methods, will treat generative functions essentially as a mathematical black box, so they fully support any model made of these building blocks.

9 September 30th, 2021

Today, McCoy Becker will present a lecture on recent research in formal verification of higher-order probabilistic programs. (There will be a lot of nLab links in this presentation.)

9.1 Formal Verification of Higher-Order PPs [SAB+19]

When we talk about formal verification of probabilistic programs, there are several techniques used in the paper, which use category theory and other formal methods to verifying probabilistic computation. But first, let's talk about motivation. What is the point of probabilistic programming, and why should we care about it at all?

McCoy sees probabilistic programming (such as what is done at his affiliated ProbComp lab) as the intersection of symbolic reasoning, data-driven learning, and statistical modeling approaches. This allows probabilistic programming to outperform either symbolic methods or machine learning models at certain tasks like online data cleaning and adaptive time-series modeling, by combining the strengths of both kinds of approaches.

In that case, where does formal verification fit in? Think about tensor shape checkers for neural network specification graphs like PyTorch and TensorFlow, which allow people to automatically check if the shapes of their tensors don't match up, saving time and money by avoiding mistakes. Formal verification of probabilistic programs works in a similar way. There are many ways in which a human programmer could make mistakes while implementing methods like importance sampling or MCMC, where they violate some assumptions of the model (e.g., trying to rejection sample a Gamma distribution on $[0, \infty)$ with a Uniform supported on $[0, 1]$).

```
(defquery Bayesian-linear-regression

 (let [f (let [s (sample (normal 0.0 3.0))
               b (sample (normal 0.0 3.0))]
           (fn [x] (+ (* s x) b)))]

   (observe (normal (f 1.0) 0.5) 2.5)
   (observe (normal (f 2.0) 0.5) 3.8)
   (observe (normal (f 3.0) 0.5) 4.5)
   (observe (normal (f 4.0) 0.5) 6.2)
   (observe (normal (f 5.0) 0.5) 8.0)

   (predict :f f)))
```

Figure 3: Using higher-order functions in Anglican to model Bayesian linear regression.

The primary means by which we map code into formal logic for verification is through *denotational semantics*, which PL researchers use to assign mathematical meaning to blocks of code. In some simple languages like the untyped lambda calculus, the denotational semantics are fairly simple. However, for automatic differentiation and probabilistic programming, you unfortunately need very complex mathematical models, like *differentiable manifolds* and *quasi-Borel measures*. This is part of the reason why the paper we are discussing has so much notation.

Going back to probabilistic programming, the core idea in general is to manipulate **computable representations of operations on probability measures**. Last lecture, we saw Gen's *generative function interface*, which is one way to model probabilistic programs, but many systems offer different tradeoffs in terms of what is efficiently representable. What's really tricky in general is how to symbolically represent joint probability distributions like the ones produced by graphical models, and this is precisely what we would like to do for verification.

However, trying to represent probabilistic programs in plain measure theory is not enough, since many probabilistic programs use *higher-order functions*, such as `f` in Fig. 3. Notice that `f` is a linear form that closes over two random variables sampled from normal distributions, so the denotational semantics of this program would be that `f` is a kind of "measure over functions." What does this even mean? It unfortunately doesn't fit in our standard models, i.e., **Meas**, the category of *measurable spaces*.[6]

9.2 Category Theory of Quasi-Borel Spaces

Category theory is "unavoidable" when thinking about denotational semantics, since it provides a lot of the necessary language to talk about universal constructions, types, and operations on them. In category theory, most computer scientists generally work in a special kind of environment called *Cartesian closed categories (CCCs)*. These satisfy three properties:

- The category $\mathcal{C}$ has a terminal object, i.e., there exists $I \in \mathrm{Ob}(\mathcal{C})$ such that for all O, there is exactly one morphism $O \to I$.
- ("Record types"). Any two objects X and Y have a product $X \times Y$.
- ("Function types"). Any two objects Y and Z have an exponential Z^Y.

The reason why we care about CCCs in programming languages is because their *internal language* is the simply typed lambda calculus, meaning that there is a Curry-Howard isomorphism between types in STLC and any Cartesian closed category (e.g., the category **Set** of sets).

However, there is one glaring problem with this mechanism. It turns out that **Meas** is not Cartesian closed! This means that while we can represent *measurable functions* as morphisms in this category, we cannot represent higher-order functions between measurable functions, since there is no exponential object.

$$\begin{aligned}&\texttt{twoUs} \equiv \texttt{let}\, u_1 = \texttt{Uniform}(0,1)\ \texttt{in}\ \texttt{let}\, u_2 = \texttt{Uniform}(0,1)\ \texttt{in}\\ &\qquad \texttt{let}\, y = u_1 \otimes u_2\ \texttt{in}\\ &\qquad \texttt{query}\ y \Rightarrow \lambda x.(\texttt{if}\ \pi_1(x) < .5 \vee \pi_2(x) > .5\ \texttt{then}\ 1\ \texttt{else}\ 0)\end{aligned}$$

Figure 4: Example of a simple joint density on $[0,1]^2$ in HPProg.

The key innovation in this paper is figuring out a way to define an STLC on top of quasi-Borel spaces (QBS), which can be used to represent higher-order functions for probabilistic programming. This is theoretically based on a variant of the *Giry monad*, which is a probability monad from **Set**, but we work through a language called *HPProg*, which is a new language for representing higher-order probabilistic programs based on the internal language of this monad.[7]

[6] This is not quite the same as a measure space! Measurable spaces are just sets equipped with a σ-algebra.

[7] There is other work on deriving a functor from **Meas** to QBS, so they are not unrelated.

Using this new language, the authors verify some simple properties on probabilistic programs using higher-order functions, and it shows a fairly eloquent way of enriching basic programs with quasi-Borel mathematical structure. They provide several examples of their language, such as the simplest example in Fig. 4. In conclusion, there are some really cool new results in giving denotational semantics to probabilistic programs, and this enables new mechanisms of verification, but the base language is still fairly new and clunky.

10 October 5th, 2021

Today, Alex Gu will present a lecture on *Dex*, a prototype array programming language.

10.1 Array Programming With Typed Indices [MRJV19]

The first observation that makes *Dex* possible is a **duality between arrays and functions**. If we had a function `f: 'a -> 'b -> 'c`, then we can define a function `g = flip f` with reversed type signature `g: 'b -> 'a -> 'c` by simply writing `g = lam x y. f y x`. Similarly, to take the transpose of a two-dimensional array `x`, we could simply write `y = transpose x`, which is a procedure corresponding to the loop `for i j. x.j.i`.

Therefore, to make typed arrays possible with generic sizes and compile-time verification of properties, Dex uses a set of finite types called `Fin n`, and the type of an array is written `i=>x`, where `i` is the finite index set, and `x` is the value type. Furthermore, this compile-time checking allows for very expressive code (similar to for-loops) that can be statically optimized on the GPU. The argument types and return types of functions are self-documenting, which also contributes to clarity. Here are a few examples, along with equivalent notation in `np.einsum` format.

```
outer :: i=>Real -> j=>Real -> i=>j=>Real
outer x y = for i j. x.i * y.j
-- 'ij<-i,j' in einsum notation

matvec :: i=>j=>Real -> j=>Real -> i=>Real
matvec x y = for i. sum (for j. x.i.j * y.j)
-- i<-ij,j

matmul :: i=>k=>Real -> k=>j=>Real -> i=>j=>Real
matmul x y = for i j. sum (for k. x.i.k * y.k.j)
-- ij<-ik,kj

trace :: i=>i=>Real -> Real
trace x = sum (for i. x.i.i)
-- <-ii

pairwiseL1 :: n=>d=>Real -> n=>n=>Real
pairwiseL1 x = for i j. sum (for k. abs (x.i.k - x.j.k))
-- (no einsum equivalent)
```

Notice that in the above examples, the for-loops did not have any specific type annotation or ranges attached to them. This is because their variable type of `Fin n` can be inferred through a Hindley-Milner algorithm, which makes it easy for the programmer to write correct, bounds-checked loops without much syntactic overhead. Also, all of these functions are polymorphic over type variables corresponding to the shape, as seen in their signatures.

There are a few other more advanced types that are supported by the array programming system. First is the notion of a *product type* of index sets, which is what happens when you flatten an two-dimensional array `m=>n=>a` to a one-dimensional array `(m,n)=>a`. Using *existential types*, you can also represent "ragged arrays" where the length of each row in the array is different. For example, `n=>(E m. m=>Real)` would be a ragged array on index set `n`.

10.2 Advanced Features of Dex

Dex also comes with a built-in effect system inspired by the Haskell `ST` monad. This allows the specification of operations like `sum` in terms of a sequential algebraic effect. Dex also provides the function `runAccum` for a subset of common effect operations that can be parallelized on the GPU. From a PL perspective, Dex also ships algebraic data types, dependent pairs, type classes, implicit arguments, and automatically synthesized arguments.

The value-dependent type system is also fairly interesting, as a user might want to specify an array type where the dimension of the array is some mathematical function of another array's dimension. This is possible using Dex's value-dependent types, as Dex separates the core language into *values* and *expressions*, with the caveat that type unification is not implemented for different variables. This means that if you have two variables both equal to the same number `a = b = x * x`, then `Fin a` and `Fin b` would be considered different types.

Finally, we briefly discuss a high-level overview of automatic differentiation in Dex. First, forward-mode AD is handled as a compile-time pass using the pointed differentiation monad, based on [Ell18]. This allows functions $f : x \mapsto y$ to be automatically transformed into a corresponding function $\mathcal{D}f : (x, \mathrm{d}f_x) \mapsto (y, \mathrm{d}f_y)$. Higher-order derivatives can also be expressed, and reverse-mode AD (which is used for efficient backpropagation in neural networks) can be cleanly implemented using a transposition operator. This produces a *calculus* of gradients and differentiation.

11 October 7th, 2021

Today, Max Snyder will present a lecture on OMeta, an object-oriented language for pattern matching, which generalizes parsing expression grammars (PEGs) and seamlessly integrates them into a host language. The intention is to make it easier to write interpreters and compilers.

11.1 Parsing Expression Grammars

OMeta is an embedded domain-specific language that offers a variant of *parsing expression grammars*, which are a way of specifying grammars in terms of strictly-ordered pattern matching rules. Parsing expression grammars allow programmers to specify formal grammars in terms of a set of non-terminal rules, which each consist of a sequence of either terminals (literals) or non-terminals. Besides sequencing, there a few other operators:

- The *choice* operator `e1 / e2` tries matching `e1`, and on failure, it matches `e2`.
- The *zero-or-more*, *one-or-more*, and *optional* operators `e*`, `e+`, and `e?` match n occurrences of the sub-expression, greedily matching as much input as possible and never backtracking.
- The *negation* operator `!e` and *lookahead* operator `&e` try to match a pattern, succeed or fail, and do not consume input.

Unlike extended-BNF grammars and parser generators based on theories like LL(k) or LR(k), PEGs do not generate symbol tables, instead having a simple correspondence with the operational semantics of a recursive descent parser. Unlike regular expressions, they have more powerful support for context-free datatypes, rather than just regular languages, but they do not backtrack.

```
1  json = { SOI ~ (object | array) ~ EOI }
2
3  object = { "{" ~ pair ~ ("," ~ pair)* ~ "}" | "{" ~ "}" }
4  pair   = { string ~ ":" ~ value }
5
6  array = { "[" ~ value ~ ("," ~ value)* ~ "]" | "[" ~ "]" }
7
8  value = { string | number | object | array | bool | null }
9
10 string  = @{ "\"" ~ inner ~ "\"" }
11 inner   = @{ (!("\"" | "\\") ~ ANY)* ~ (escape ~ inner)? }
12 escape  = @{ "\\" ~ ("\"" | "\\" | "/" | "b" | "f" | "n" | "r" | "t" | unicode) }
13 unicode = @{ "u" ~ ASCII_HEX_DIGIT{4} }
14
15 number = @{ "-"? ~ int ~ ("." ~ ASCII_DIGIT+ ~ exp? | exp)? }
16 int    = @{ "0" | ASCII_NONZERO_DIGIT ~ ASCII_DIGIT* }
17 exp    = @{ ("E" | "e") ~ ("+" | "-")? ~ ASCII_DIGIT+ }
18
19 bool = { "true" | "false" }
20
21 null = { "null" }
22
23 WHITESPACE = _{ " " | "\t" | "\r" | "\n" }
```

The above code block (source) shows an example of a full, specification-compliant parsing expression grammar for JSON, written using the Pest library in Rust. Notice how the PEG is written as a simple syntax consisting of a number of definitions for non-terminal rules.

Note how the Pest syntax is slightly different from the mathematical specification above, as sequence requires the `~` operator, and choice is expressed with the `|` operator. This shows a general trend in the parser community, where different implementers will adapt the syntax of PEGs to the language of their choice. There are other features supported by some PEG libraries, like operator precedence, left recursion, and result transformations, at the cost of some complexity.

11.2 OMeta: PEGs on Arbitrary Datatypes [WP07]

OMeta is a slight variant on traditional parsing expression grammars with very similar operational semantics. The key difference is that instead of matching on character sequences, like traditional PEGs, OMeta can match on sequences of arbitrary host language objects. The paper specifies the formal semantics in mathematical notation, which we go through briefly.

Our first live example is an interpreter for a simple programming language, which we will build using the paper's original *OMeta/JS* system. Unlike traditional PEG systems that are laser-focused on the parsing task, OMeta tries to offer advanced features under the belief that many programming language problems (like interpretation, optimization passes, and compilation) can be ergonomically written as generalizations of pattern matching.

```
ometa Calculator {
  token :t
    = space* char:c ?(t == c)
  digit
    = ^digit:d -> d.digitValue(),
  number
    = number:n digit:d -> ((n * 10) + d)
    | digit,
  fact 0 = -> 1,
  fact :n = fact(n-1):m -> (n * m),
  expr
    = "(" expr:a "+" expr:b ")" -> (a + b)
    = "(" expr:a "-" expr:b ")" -> (a - b)
    = "(" expr:a "*" expr:b ")" -> (a * b)
    = "(" expr:a "/" expr:b ")" -> (a / b)
    = "(" expr:e ")" "!" fact(e):f -> f
    | space* expr
    | number
}

Calculator.matchAll("123", "number"); /* RESULT: 123 */
Calculator.matchAll("(100+550)", "expr"); /* RESULT: 650 */
Calculator.matchAll("(100)!", "expr"); /* RESULT: 650 */
```

A couple notes about the code above. The `letter` and `digit` rules are a convenience built into every OMeta/JS grammar by default. The `^` syntax denotes inheritance from the base rule. Also, `token :t` is a parameterized rule that has special meaning in the pattern syntax, where `"("` cor-

responds to `token('(')`. Notice how we very cleanly implemented a recursive, pattern-matched function in *fact* within the grammar itself!

Here's another cool example of how OMeta can be used to construct modular programs that build on each other using inheritance. Max creates a new `HexCalculator` parser class that how works in hexadecimal.

```
ometa HexCalculator <: Calculator {
  digit
    = ^digit:d -> d.digitValue()
    | 'A' -> 10
    | 'B' -> 11
    | 'C' -> 12
    | 'D' -> 13
    | 'E' -> 14
    | 'F' -> 15,
  number
    = number:n digit:d -> ((n * 16) + d)
    | digit
}

HexCalculator.matchAll("(AAA-123)", "expr"); /* RESULT: 2439 */
```

Finally, Max provides a more complicated calculator example using a higher-order `vector :t` rule. The goal here is that "DSLs should be made by not PL people," in other words, enabling more approaches in the creative process based on pattern matching.

12 October 12th, 2021

Today, I (Eric Zhang) presented a lecture on DiffTaichi, a differentiable programming language for accelerated graphics and physical simulation, which is embedded in Python [HAL+19].

13 October 14th, 2021

Today, Yafah Edelman presents a lecture on PyTorch, a very popular deep learning framework for Python, with dynamic dataflow graphs, eager execution, and an imperative style.

13.1 Design of PyTorch [PGM+19]

To understand the design of PyTorch, it's first useful to understand the alternative design of other frameworks like TensorFlow, which have a static computation graph.

In TensorFlow, you define symbolic variables called *tensors*, which are multi-dimensional arrays that are stored on the GPU. These `tf.Tensor` and `tf.Variable` objects are symbolically manipulated by mathematical operators in the `tf` namespace, like `tf.add` and `tf.matmul`. As this is done, you create a computation graph for your entire program, which you then statically run with an evaluator on the GPU. This design allows for very efficient, flexible CPU/GPU evaluation and automatic reverse-mode differentiation.

However, PyTorch takes a different approach. Unlike in TensorFlow, where computation graphs must be entirely static, PyTorch instead takes a more imperative approach, where the computation graph is reference-counted[8] and built/evaluated dynamically while the actual Python code is being run. This breaks PyTorch programs down into two interlocking parts:

- **Control flow**, handled by vanilla Python code (e.g., loops and `if` statements).
- **Data flow**, asynchronously handled by the efficient CPU/GPU backend and interacting directly with control flow.

Eager evaluation has been tremendously influential since PyTorch was introduced, and it was even made the default mode of operation in TensorFlow 2.0. The reason is that eager evaluation makes tensors behave more imperatively, which satisfies the design goal of being more *Pythonic* — easier for lay programmers to understand and use. It also accelerates machine learning research, since it is more flexible with less code, and you can insert `print` statements.

```
import torch
import torch.nn as nn

x, y = load_data()
model = nn.Sequential(nn.Linear(4, 8), nn.ReLU(), nn.Linear(8, 1))

loss_fn = nn.BCEWithLogitsLoss(reduction="sum")
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)

for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)
    if not t % 100:
        print(t, loss.item())
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

[8] Tensor memory on the GPU is automatically freed when the reference count reaches zero.

In the above example, we train a simple two-layer fully connected neural network with stochastic gradient descent, on a supervised binary classification task. On every 100 iterations of gradient descent, we print out the loss of the current model in training.

13.2 Tracing in PyTorch

Now we talk about a more complex topic, which is how PyTorch implements *tracing*. This provides insight into how the control flow and data flow of the library interact with each other.

```
import torch
import torch.nn as nn

class TracingExample(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(4, 4)

    def forward(self, x):
        m = self.linear(x)
        if m.sum() > 0:
            print("m was Positive")
        else:
            print("m was Negative")
            m = -m
        res = torch.tanh(m)
        return res, res

example = TracingExample()
x = torch.randn(1, 4)

traced_eample = torch.jit.trace(example, (x,), check_trace=False)
print(traced_example.code)

# m was Positive
# def forward(self, x: Tensor) -> Tuple[Tensor, Tensor]:
#   m = torch.tanh((self.linear).forward(x, ))
#   return (m, m)

# m was Negative
# def forward(self, x: Tensor) -> Tuple[Tensor, Tensor]:
#   m = torch.neg((self.linear).forward(x, ))
#   _0 = torch.tanh(m)
#   return (_0, _0)
```

There are two possible traces in this above example, based on whether m had a sum that was positive or negative. Notice how arbitrary Python control flow, which is run on the CPU (and cannot be placed on a hardware accelerator) directly changes the data flow that is eagerly evaluated by the GPU. This means that PyTorch needs to dynamically generate its dataflow graph, which adds flexibility at the cost of some overhead (as shown by `torch.jit.trace`).

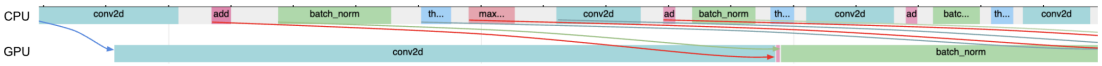

Figure 1: A trace of the first few operators of Resnet-50. The top row depicts the execution of the control flow running on the host CPU. The gray areas are Python code executed by its interpreter. The colored areas correspond to the work done on the host CPU to queue various operators (convolution, batch normalization, and so on). The bottom row shows the corresponding execution of those operators on the GPU. The arrows pair the two events in time.

Figure 5: A PyTorch trace, showing the asynchronous evaluation of data flow.

It's important to note that ML researchers today care deeply about efficiency, which is different from research in the past. This is because many of the largest innovations in machine learning require tremendous amounts of computational power and money, so people spend much more time on speed, optimization, hyperparameter tuning, and high-performance computing tasks like data parallelism and distributed training. These engineering challenges are what make PyTorch such a massive and complex framework that enables everything from small research experiments to industrial language models.

The importance of speed also makes companies like Google and Nvidia, who make hardware accelerators, very relevant to PyTorch's development (see the MLPerf benchmark). Despite the fact that PyTorch is written in Python, all of the most important bottlenecks in numerical performance are handled by optimized C++ code and GPU kernels, which makes the overall execution extremely efficient. It also enables operations like scheduling similar operations at temporally local points in GPU evaluation, as shown in Fig. 5.

14 October 19th, 2021

Today, we had a guest lecture from Aaron Bembenek on Formulog [BGC20]. This is a project that I've worked on before in a research internship with Aaron.

15 October 21st, 2021

Today, Kevin Zhang will present a lecture on using Datalog for large-scale program analysis, specifically looking in depth at the Soufflé framework.

15.1 Introduction to Datalog [JSS16]

Datalog is a logic programming language originally created for database query applications. It is fairly simple, and programs are written as a collection of rules. They start with an *extensional database (EDB)* of facts specified at the beginning of execution, and they derive an *intensional database (IDB)* by repeatedly evaluating Horn clauses until reaching a fixed point.

How does this translate to program analysis? Usually, we have an input program written in some language, and we write an extractor in a general-purpose programming language that generates relations from the code. Then, the Datalog runtime uses these relations as input to derive static analysis from applying Horn clauses to these relations until reaching a fixed point. Here's how you might write a simple points-to analysis for a small LLVM-based language.

```
.type var <: symbol
.type obj <: symbol
.type field <: symbol

// -- inputs --
.decl assign( a:var, b:var )
.decl new( v:var, o:obj )
.decl ld( a:var, b:var, f:field )
.decl st( a:var, f:field, b:var )

// -- facts --
assign("v1","v2").

new("v1","h1").
new("v2","h2").
new("v3","h3").

st("v1","f","v3").
ld("v4","v1","f").

// -- analysis --
.decl alias( a:var, b:var ) output
alias(X,X) :- assign(X,_).
alias(X,X) :- assign(_,X).
alias(X,Y) :- assign(X,Y).
alias(X,Y) :- ld(X,A,F), alias(A,B), st(B,F,Y).

.decl pointsTo( a:var, o:obj )
.output pointsTo
pointsTo(X,Y) :- new(X,Y).
pointsTo(X,Y) :- alias(X,Z), pointsTo(Z,Y).
```

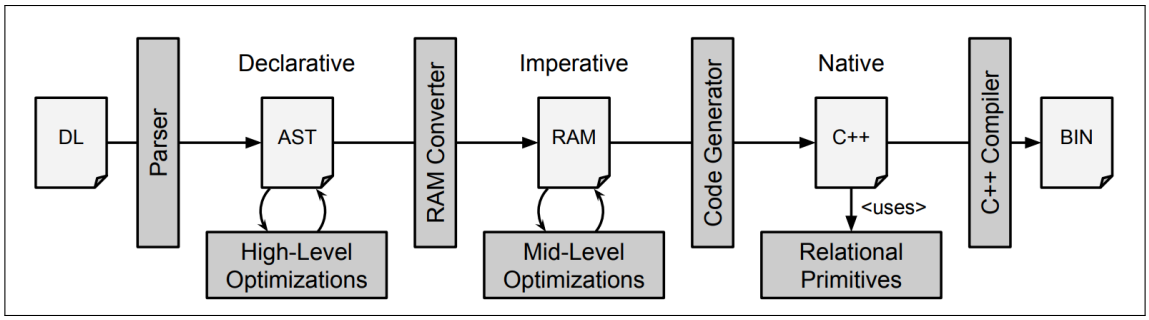

Figure 6: Staged compilation pipeline for Datalog in Soufflé.

A fairly common technique is to have the extractor encode the control-flow graph (CFG) of a program in terms of input relations. Then, the Datalog program can recursively iterate over these control-flow graphs and encode many types of static analysis, such as dataflow analysis.

15.2 Staged Datalog Compilation in Soufflé [SJSW16]

Soufflé's approach to evaluating Datalog allows you to either interpret Datalog using an efficient C++ interpreter based on parallel B-trees, or using a staged variant that requires a lot of compilation time, but it allows for more optimization and is faster during runtime.

Generally, Datalog compilation in Soufflé comes in several lowering steps, shown in Fig. 6. First, the program is analyzed for data dependencies to compute strongly-connected components, and assumptions like stratified negation are verified. Then, the Datalog program is translated into a mid-level *relational algebra machine (RAM)* representation, using the well-known *semi-naive evaluation* strategy that efficiently considers new facts as they arrive. This abstract relational machine IR can either be interpreted by the Datalog runtime or translated directly into C++ code.

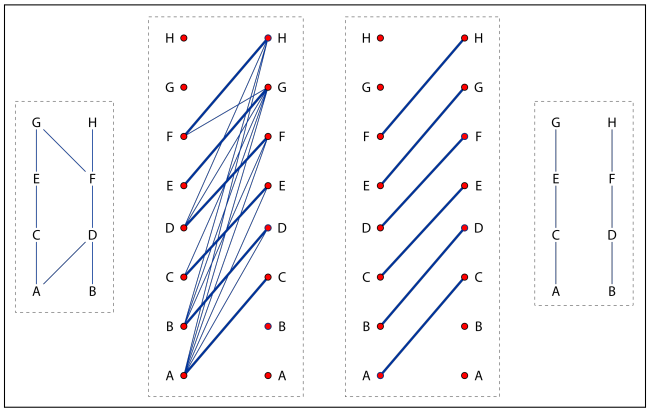

Figure 7: Bipartite graph construction used in computing minimum index sets.

There are a couple of interesting optimizations in this paper at all levels of compilation. When compiling the mid-level RAM representation, we often want to determine the minimum set of indices that is required to represent all of the *index searches* that are made on the database. This is equivalent to a minimum chain cover problem on the partially-ordered set of searches (each a set of fields to index on), which can be computed in $O(m\sqrt{n})$ time by reducing to bipartite matching (Hopcroft-Karp) using the same construction as in the proof of Dilworth's theorem. This algorithmic gadget is visualized in Fig. 7.

At the low-level interpretation side, Soufflé uses a variant of B-trees to support very fast parallel execution, since large-scale program analyses are often run on dedicated machines with many CPU processors. Originally, Soufflé used a simple read-write lock on top of its indexes, but this is not very scalable because it starves writers. Instead, their current specialized data structure is detailed in [JSZS19], which uses an atomic *seqlock* with upgradable read guards.

Finally, there is an optimization in Soufflé that automatically detects *equivalence relations* and uses a union-find data structure to compute their fixed points, which is much faster than going through the entire general-purpose Datalog evaluation pipeline. There's also an engineering trick where Soufflé generalizes code specialized with compile-time templates, which allows their recursive comparator to be optimized well by GCC.

Generally, performance of Soufflé is quite impressive given the size of the input relations, even being able to handle things as large as the entirety of OpenJDK. It's also being used in production at several companies, like Galois (for security research), and it's also currently being added as an alternative backend for Polonius, the Rust borrow checker.

16 October 26th, 2021

Today, Tyler Holloway will be presenting on *egg*, a library for fast and extensible equality saturation.

16.1 Extensible Equality Saturation [WNW+21]

We start by describing *term rewriting*, a common technique in optimization where compilers find specific patterns within an expression, then iteratively replacing them with equivalent but simpler terms. However, rewriting is a destructive one-way process, so depending on which order you rewrite rules, you may not find the simplest final result.

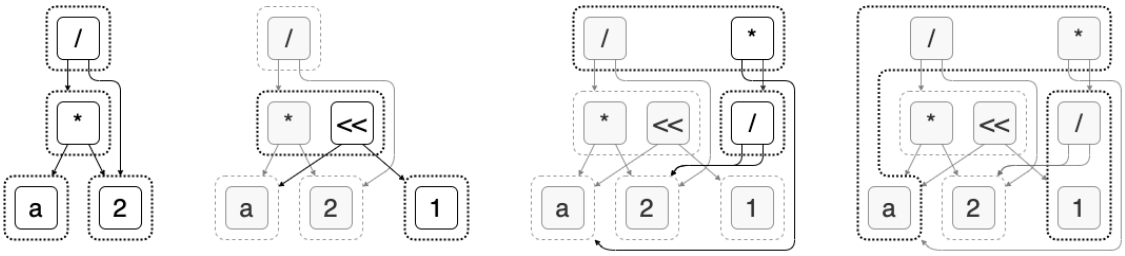

Figure 8: E-graph for an arithmetic language as it undergoes a series of rewrites.

The motivation for *equality graphs (e-graphs)* is to describe a data structure that lets you represent a large (exponentially sized) set of programs equivalent to the original term, by applying a set of simplification rules until reaching a fixed point (*saturation*). Then, the optimizer extracts an the best term term from the final equality graph, which represents all possible equivalent programs under the optimizing rewrites, based on a user-specified cost function.

In the above example, Fig. 8, the small solid boxes with operators represent *e-nodes*, while the dashed boxes represent *e-clasess*, which are equivalence classes of e-nodes. Notice that on each successive application of a rewrite rule, there are additional nodes that allow us to represent additional equivalent expressions to `(a * 2) / 2`. In each step, our rewrite rules only add information to the graph, never deleting information or removing potential term equivalences.

Currently, equality saturation algorithms based on e-graphs are used in most modern theorem proving and SMT systems. However, they tend to be ad hoc and require domain-specific extensions to the e-graph, since they are difficult to run efficiently. Today's paper, *egg*, provides a Rust library that supports general-purpose e-graphs for equality saturation, with two innovations for efficiency:

- **Deferred invariant maintenance:** The egg library adds a *rebuild* function to the e-graph, which recomputes the graph by deduplicating nodes and storing the canonical representation of each e-node. This makes it so that we only need to check invariants after each *iteration* of equality saturation before reaching a fixed point, since we batch all *merge* operations until all of work for a single pattern search is completed.

- **E-class analysis:** One common extra analysis added to e-graphs is constant folding, where we eliminate constant subexpressions like `(2 + 4) * 5` that can be evaluated at compile time. In most previous implementations, constant folding was an ad hoc optimization bolted on to the e-graph, but egg makes this explicit by adding an analysis system, which annotates each e-class with facts drawn from a semilattice domain. Whenever two e-classes are merged, their associated analysis values are joined in the semilattice.

For two good resources about e-graphs, please see the rustdoc, as well as `egraphs_example.ipynb`, a minimal Python implementation with an associated visual tutorial.

16.2 E-Graphs Discussion

There are some discussion topics of further interest related to e-graphs:

- Can we use e-graphs to develop a kind of alternative optimizer algorithm for LLVM? Typically, LLVM applies a series of optimization passes, and each pass loses information, so the problem of *phase ordering* is important.
- How do e-graphs compare to normalizing rewrite rules in other systems also based on pattern-matching, such as OMeta?
- How do e-graphs compare to the actual data structures used by theorem provers? It seems like they are most similar to the unification algorithms in symbolic reasoning systems, but the specific data structure was not very standardized before this paper.
- Can you use e-graphs for extending or augmenting program synthesis techniques, such as the example in the paper where *Szalinski* [NWA+20], using equality saturation on e-graphs, is able to synthesize simpler programs that represent a constructive solid geometry.
- What about approximate equality in applications like floating-point geometric programs, where you may want to simplify a shape within a volume difference of ϵ?

17 October 28th, 2021

Today, Jack Boettcher will present a lecture on Grammar VAEs, a specialized neural architecture for generative modeling problems over strings of a context-free grammar.

17.1 Grammar Variational Autoencoders [KPHL17]

Recall that *variational autoencoders* are a form of generative model that combines an encoder network, modeling $p_\theta(\mathbf{z} \mid \mathbf{x})$, with a decoder network, modeling $q_\theta(\mathbf{x} \mid \mathbf{z})$. Here, $\mathbf{x}$ is a data vector being modeled, and $\mathbf{z}$ is a vector in the latent space. We set a prior on $\mathbf{z}$ to be multivariate Gaussian with unit variance, and then we train the entire neural network with variational Bayes, optimizing the ELBO. This produces an implicit generative model that allows us to sample $\mathbf{z}$ from the latent space and synthesize new points in the data distribution.

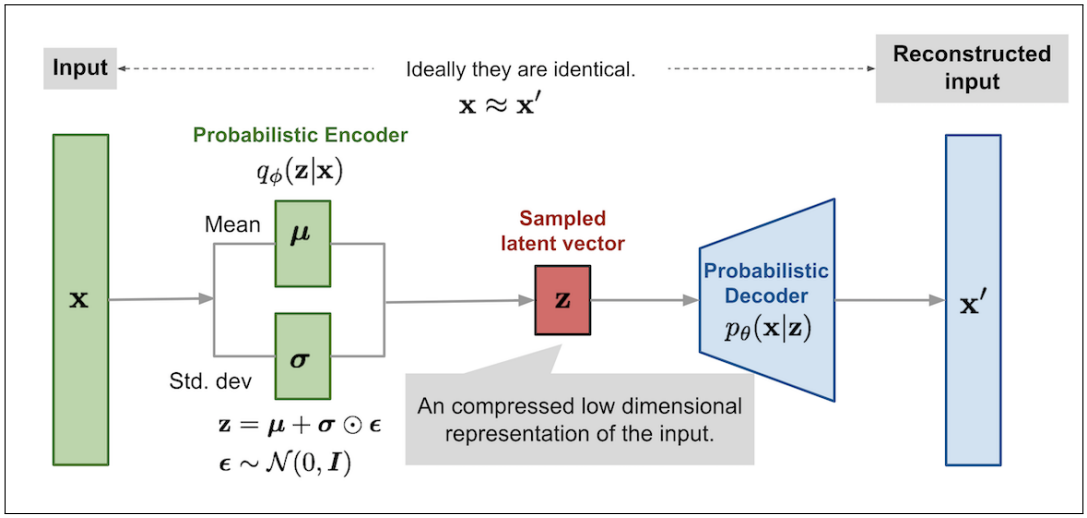

Figure 9: Architecture diagram of a typical VAE, with bottleneck in the latent space.

A layer-by-layer diagram of a variational autoencoder, as typically implemented in deep learning frameworks, is shown in Fig. 9. To learn more about the mathematics behind this model, see the highly-cited original paper ([KW13]), or view Stefano Ermon's course notes on generative models.

However, standard variational autoencoders for character-level tasks work well for unstructured languages, but they do not work well for structured data that can be modeled by a context-free grammar. The example in this paper is SMILES, a system for representing molecular structures as strings of text. Many times, a character-level VAE trained on SMILES would generate text that does not follow the grammar. Grammar VAEs make two improvements:

- **Encoder:** Instead of embedding each individual character, the encoder instead generates the parse tree, then passes as input a one-hot encoding of each *production rule* in a preorder traversal of the parse tree.
- **Decoder:** A fixed-size matrix is sampled from the variational decoder, then it is traversed sequentially by index. Each index has an associated location in the intermediate parse tree, and we mask out those matrix elements that do not make sense in the context. Then, we sample a production rule or terminal according to the output probability distribution.

Together, these improvements are generally able to increase accuracy by reducing the number of samples that are malformed, using the grammar as a guide to efficient generative modeling.

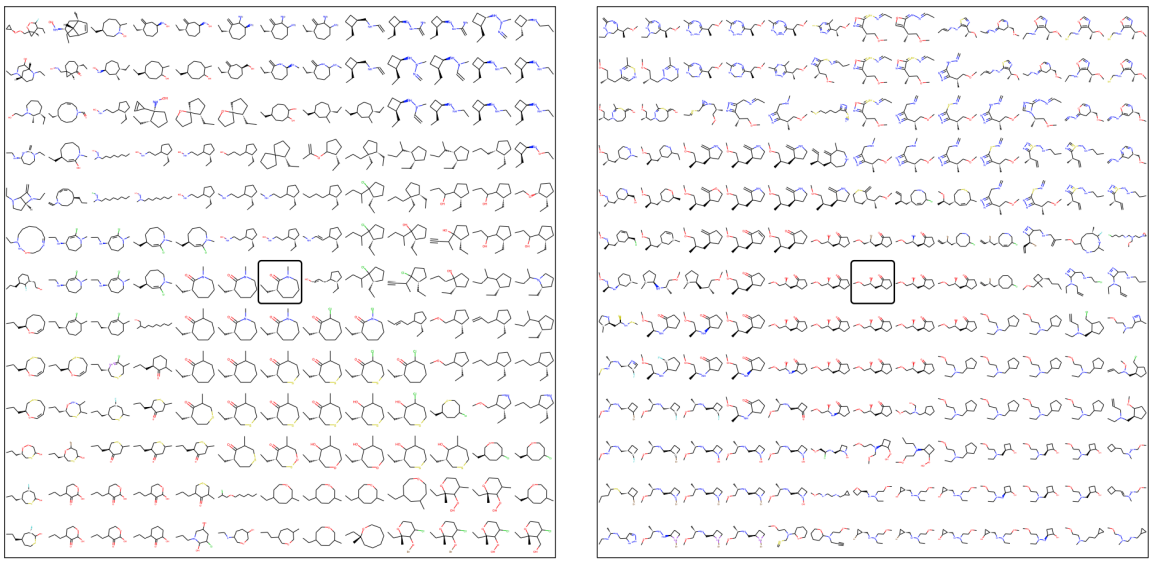

Figure 10: Cross-sections of the latent space of a GVAE trained on SMILES molecules.

17.2 GVAE Results and Analysis

The results of grammar variational autoencoders visually show a fair amount of promise, compared to their equivalent character-level baseline. For example, we can do the standard demonstration of interpolating between points in the latent space. In Fig. 10, we can see some visual demonstrations of interpolation in the latent space of SMILES parse trees.

Another demonstration is the ability to fit a symbolic arithmetic expression, which is trained based on a loss that is actually dependent on the mean squared-error between the function result and expected data points. We can find a global optimum for both character-level and grammar variational autoencoders, using gradient-free Bayesian optimization (BO) methods over the latent space, which is a probabilistic method for optimizing black-box functions.

Using a similar Bayesian optimization method, the authors also provide a larger demonstration where they optimize molecules for certain chemical properties using their GVAE model. This allows them to do some form of automated drug discovery, and they benchmark both models. The best molecules found by the GVAE have significantly better *LogP* score compared to the CVAE.

18 November 4th, 2021

Today, we had a guest lecture from Thomas Bourgeat, who described work on hardware descriptor languages (HDLs), processor pipelining and parallelism, Bluespec (rule-based synthesis) [H+00], and formal verification for correctness and liveness properties in hardware [BPCC20].

19 November 9th, 2021

Today, we have a guest lecture from Alexey Radul on automatic differentiation.

19.1 Introduction to Automatic Differentiation [BPRS18]

Generally, the goal of automatic differentiation is to algorithmically evaluate derivatives of composite functions, expressed via code, in a way that allows for complex functions from composing primitives. This allows programmers to compute derivatives for expressions that would be intractable to symbolically differentiate, due to a combinatorial explosion in size from repeatedly applying the product and chain rules. There are two main modes of AD:

- **Forward-mode:** Given scalar x and a vector function $\mathbf{y} = f(x)$, compute $f'(x)$. This is conceptually simpler and has a nice mathematical interpretation.
- **Reverse-mode:** Given a scalar function $y = f(\mathbf{x})$, compute the gradient $\partial y/\partial \mathbf{x}$. This is more complex, but it is also more useful in machine learning for training neural networks via gradient descent (also known as *backpropagation*).

In forward-mode automatic differentiation, for each smooth function $f : X \to Y$, we transform it to the differential given by $\mathcal{D}f : TX \to TY$, a covariant functor sending differentiable manifolds to their tangent bundles. This can be done automatically by mapping each programming language operation[9] into its corresponding differentiation rule. For example,

$$\begin{aligned}\mathcal{D}(f+g)(x,v) &= \mathcal{D}f(x,v) + \mathcal{D}g(x,v)\\ \mathcal{D}(fg)(x,v) &= \mathcal{D}f(x,v)\cdot g(x) + f(x)\cdot \mathcal{D}g(x,v)\\ \mathcal{D}(f\circ g)(x,v) &= \mathcal{D}f(g(x), \mathcal{D}g(x,v)).\end{aligned}$$

In a general-purpose programming language, you can extend the language with a forward-mode AD system by overloading each mathematical operator with its corresponding differential variant operating on the tangent bundle. To do computation, you simply pass $\langle x, x'\rangle$ to the differential of any function $f(x)$, where x' is the perturbation direction that you want to query.

19.2 Reverse-Mode Automatic Differentiation

Although forward-mode automatic differentiation is conceptually simple, it is intractable for many deep learning problems, where we have a complex function f_θ that sends a high-dimensional input vector into a scalar objective value. In order to compute $\partial f/\partial\theta$, the gradients of f with respect to its parameters θ, we would need to run one forward-mode differentiation pass for *each* dimension in the input vector, which is very inefficient.

Luckily, there is an alternative, much more efficient approach known as *reverse-mode* automatic differentiation, which only requires a single pass to find gradients with respect to a scalar objective value. Similar to how forward-mode differentiation pushes elements of the tangent bundle forward, the reverse-mode variant pulls cotangent vectors back, through the corresponding contravariant functor $\mathcal{D}^*f : T^*Y \to T^*X$ to the cotangent bundle.

However, reverse-mode automatic differentiation is much more subtle. The reason is that, while your primal computations in AD still need to run forward regardless, the actual derivatives are propagated backward, forming a half-covariant and half-contravariant structure. This has the following, mildly unfortunate implications:

[9] We'll assume static single assignment form (SSA) for the rest of the lecture.

- Each intermediate value in the primal computation of $f(\mathbf{x})$ needs to be stored up to the entire duration of the computation. Its lifetime needs to extend past scopes, closures, and other control flow, until the reverse pass is finished. This is roughly equivalent to doing forward-mode differentiation, reordering the tangent computations to the end (extending lifetimes), then taking the "transpose" of each operation.
- The memory requirements of reverse-mode AD are linearly proportional to the runtime requirements of forward-mode AD.
- The differentiation itself requires a *tape* to store intermediate gradients and requires reversing the the actual control flow of the language, including conditional `if` expressions and `for` loops. This is troublesome and requires careful implementation.
- Each forward-mode computation, which can be expressed as a linear operator between tangent bundles $TX \to TY$, is roughly "transposed" into its dual, which turns dense *gather* operations on arrays and structs into sparse *scatter* operations across lots of data.

There are active topics of research in automatic differentiation, particularly related to how to make reverse-mode AD simpler. For example, Alexey Radul works on Dex [MRJV19], which we discussed in an earlier lecture. There are interesting questions around how to make the tape more memory-efficient, particularly for operations like differentiating through array indexing, which requires many sparse updates if not optimized carefully.

Alexey also answers some additional, miscellaneous questions from students about various topics related to automatic differentiation:

- **How do you do mixed-mode AD?** Most mixed-mode automatic differentiation systems are roughly equivalent to running forward-mode AD on one part of the computation to generate tangent vectors, then running reverse-mode automatic differentiation on the rest.
- **What about source code transformations?** In general, reverse-mode automatic differentiation always requires some kind of tape to reflect lifetimes of the primal computation. However, some languages push this computation back to compiler code generation, which can make these more efficient. The underlying mechanism is still the same, though.
- **What is checkpointing?** Suppose that you have a really long function that does not use much memory, being a relatively narrow but lengthy computation. Instead of storing all of the potentially-quadratic memory for this function at once, you can instead store *checkpoints*, then only rerun small chunks of the computation at a time, as gradients are pulled back.

Another interesting analogy is that automatic differentiation is similar to *taint analysis* in systems security. At least superficially, these binary-level taint tracking systems attach tags to data, similar to how automatic differentiation attaches gradients to operands.

20 November 11th, 2021

Today, Ana will present a lecture on educational tools in programming languages. First, we discuss the *Calculational Style of Programming*, going into a case study on designing a programming environment for teaching formal logic, assisted by an automated theorem prover [CD14]. Next, we discuss *PEST*, a language (similar to Dafny) for teaching formal verification in universities, with a very basic type system and symbolic execution for inference [DCGG11].

21 November 16th, 2021

Today, Dougal Maclaurin, an invited guest lecturer from Google and Harvard PhD alumnus, will give a presentation on the Dex array programming language.

21.1 History of Dex [MRJV19]

Dougal's first experience with domain-specific array programming languages was when working on the Python Autograd library in 2013, as one of the original authors. After that, he spent some time working in physics, at a biotech startup, and then on JAX at Google. However, adding additional language features to JAX quickly became difficult, as doing arithmetic on symbolic expressions or branches could quickly explode. This problem motivates two directions:

- Adding AD and GPU-style parallelism to existing language implementations, such as in libraries for Julia [IEF+19] and Swift [SS21].
- Creating a new domain-specific language for array programming, i.e., Dex.

Aside from the language design space, we can also think about data representations. Broadly speaking, there are two models of data used in the field of data science.

- One is the "MATLAB model" that first originated in Iverson's APL language, which represents data in terms of dense, rectangular arrays of scalars. This was cloned in NumPy, Julia, TensorFlow, PyTorch, and so on. It tends to be more imperative.
- In contrast, the other is the "SQL model" that originates in Codd's 1969 relational algebra paper [Cod02]. This was first cloned as a data modeled in SQL, R, and later Pandas. Dougal sees the success of SQL as an indication of the power of the relational model, since despite all of SQL's shortcomings as a language, it is still immensely popular.

Both models have advantages and disadvantages, but they typically are used for different purposes. Although optimizing underlying SQL engine is of interest to database and systems researchers, there is also a lot of potential in language design for better database frontends. Bringing the best parts of these two data models together was a major motivation for Dex.

21.2 Survey of Unique Features in Dex

In Dex, arrays are seen as functional objects, which represent the reification of eager function evaluation over a finite index set. This means that you can put any finite integer type on the left side of an array `=>` symbol, such as `(Fin 5)`, `(Either (Fin 3) (Fin 4))`, and `(Fin 3) * (Fin 4)`. The index sets are the analogy to a primary key in SQL, but the arrays in Dex are flexible enough to support arbitrary value types and structured, curried indices.

Another advantage of Dex's table type is a natural hierarchy of structure. Dex can represent statically sized arrays, dynamic rectangular arrays, structured ragged arrays, general ragged arrays, and even jagged arrays in its type system. Because of its flexible generic table system, you can also implement relational / dataframe programming in Dex.

Dougal also makes a point about the utility of algebraic data types in scientific computing. In medical and scientific datasets, you often want to represent data that has categorical failures, as well as complex spaces. Representing this in terms of a single `null` value or string names for missing data is suboptimal and messy, and algebraic data types can help with this. Dex supports *non-recursive* ADTs, which can be statically lowered to tuple types using a tagged union. However,

it does not support recursive ADTs similar to Haskell's `List`. See the Dex raytrace example for a demonstration of this. A current proposed research topic in Dex is figuring out how to model recursive data, which is useful for structures like bounding volume hierarchies.

Next, we discuss parallelism in Dex. There are essentially three types of loops. By default, `for` loops are "embarrassingly parallel" when the function that is called is pure and does not have data dependencies. On the other hand, using the `withState` effect, we can reproduce serial execution. Between these two extremes is monoidal accumulation, using the `withAccum` effect. This allows the compiler to reassociate accumulation into a tree restructure to enable efficient parallelization.[10]

```
linearize : [VectorSpace a, VectorSpace b] (a -> b) -> a -> (b & a -o b)
transpose : [VectorSpace a, VectorSpace b] (a -o b) -> (b -o a)
```

Finally, we give a brief overview of the automatic differentiation setup in Dex. The entire API is specified in terms of two functions described above. The first one, `linearize`, does the forward pass of AD and stores information about the intermediate calculations. The second function, `transpose`, essentially reverses the linear function to replicate reverse-mode AD.

What's interesting about linear functionals `a -o b` in Dex is actually being specified in terms of a *linear type system* [Wad90], such as the ownership model used by Rust. This allows Dex to support curried bilinear forms like `multiply : (a -o (a -o b))`, but while `multiply x 3` would be linear, the evaluation `multiply x x` would not, since x is reused twice.

Another question is how to achieve higher-order automatic differentiation. One way is to simply repeatedly apply first-order AD as described above, which still achieves the same asymptotic time complexity, but it does some duplicate work. Having an explicit higher-order API for automatic differentiation that avoids this work duplication would be an interesting future research topic.

Dex is an active, ongoing research project, and they are active on GitHub. The project welcomes outside involvement and supports student research interns (Google Brain).

[10] This is not an entirely safe optimization, since finite precision floating-point math is *not* associative. However, the Dex compiler developers thought that this was an appropriate tradeoff for enabling parallelization. A more powerful compiler might give the user more control over associative execution order.

References

[AR17] Nada Amin and Tiark Rompf. Lms-verify: abstraction without regret for verified systems programming. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages - POPL 2017*, pages 859–873, New York, New York, USA, jan 2017. ACM Press, ACM Press.

[AS96] Harold Abelson and Gerald Jay Sussman. *Structure and interpretation of computer programs*. The MIT Press, 1996.

[AT16] Nada Amin and Ross Tate. Java and scala's type systems are unsound: The existential crisis of null pointers. *Acm Sigplan Notices*, 51(10):838–848, 2016.

[BCD+05] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. Boogie: A modular reusable verifier for object-oriented programs. In *International Symposium on Formal Methods for Components and Objects*, pages 364–387. Springer, 2005.

[BGC20] Aaron Bembenek, Michael Greenberg, and Stephen Chong. Formulog: Datalog for smt-based static analysis. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–31, 2020.

[BPCC20] Thomas Bourgeat, Clément Pit-Claudel, and Adam Chlipala. The essence of bluespec: a core language for rule-based hardware design. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 243–257, 2020.

[BPRS18] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of machine learning research*, 18, 2018.

[Byr09] William E. Byrd. Relational programming in minikanren: Techniques, applications, and implementations. phd, Indiana University, 2009.

[CD14] Dipak L Chaudhari and Om Damani. Automated theorem prover assisted program calculations. In *International Conference on Integrated Formal Methods*, pages 205–220. Springer, 2014.

[CDE+08] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[Chl16] Adam Chlipala. Ur/web. *Communications of the ACM*, 59(8):93–100, jul 2016.

[CKK+12] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c. In *International conference on software engineering and formal methods*, pages 233–247. Springer, 2012.

[Cod02] Edgar F Codd. A relational model of data for large shared data banks. In *Software pioneers*, pages 263–294. Springer, 2002.

[CTSLM19] Marco F. Cusumano-Towner, Feras A. Saad, Alexander K. Lew, and Vikash K. Mansinghka. Gen: A general-purpose probabilistic programming system with programmable inference. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 221–236, New York, NY, USA, 2019. ACM, ACM.

[DCGG11] Guido De Caso, Diego Garbervetsky, and Daniel Gorín. Pest: from the lab to the classroom. In *Proceedings of the 1st Workshop on Developing Tools as Plug-ins*, pages 5–8, 2011.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[Ell18] Conal Elliott. The simple essence of automatic differentiation. *Proceedings of the ACM on Programming Languages*, 2(ICFP):1–29, 2018.

[ERSLT17] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Joshua B Tenenbaum. Learning to infer graphics programs from hand-drawn images. *arXiv preprint arXiv:1707.09627*, 2017.

[H+00] James C Hoe et al. Hardware synthesis from term rewriting systems. In *VLSI: Systems on a chip*, pages 595–619. Springer, 2000.

[HAL+19] Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, and Frédo Durand. Difftaichi: Differentiable programming for physical simulation. *arXiv preprint arXiv:1910.00935*, 2019.

[HF13] Jason Hemann and Daniel P. Friedman. μkanren: A minimal functional core for relational programming. In *Scheme Workshop*, 2013.

[IEF+19] Mike Innes, Alan Edelman, Keno Fischer, Chris Rackauckas, Elliot Saba, Viral B Shah, and Will Tebbutt. A differentiable programming system to bridge machine learning and scientific computing. *arXiv preprint arXiv:1907.07587*, 2019.

[JSS16] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. Soufflé: On synthesis of program analyzers. In *International Conference on Computer Aided Verification*, pages 422–430. Springer, 2016.

[JSZS19] Herbert Jordan, Pavle Subotić, David Zhao, and Bernhard Scholz. A specialized b-tree for concurrent datalog evaluation. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 327–339, 2019.

[KPHL17] Matt J Kusner, Brooks Paige, and José Miguel Hernández-Lobato. Grammar variational autoencoder. In *International Conference on Machine Learning*, pages 1945–1954. PMLR, 2017.

[KS16] Daniel Kroening and Ofer Strichman. *Decision procedures*. Springer, 2016.

[KW13] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[Lei10a] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 348–370. Springer, 2010.

[Lei10b] K. Rustan M. Leino. *Dafny: an automatic program verifier for functional correctness*, volume 6355 of *Lecture notes in computer science*, pages 348–370. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[MRJV19] Dougal Maclaurin, Alexey Radul, Matthew J Johnson, and Dimitrios Vytiniotis. Dex: array programming with typed indices. *NeurIPS Program Transformations*, 2019.

[NM88] Gopalan Nadathur and Dale Miller. An overview of lambda-Prolog. *University of Pennsylvania Technical Reports*, 1988.

[NWA+20] Chandrakana Nandi, Max Willsey, Adam Anderson, James R Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. Synthesizing structured cad models with equality saturation and inverse transformations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 31–44, 2020.

[PGM+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*, page 8024–8035. Curran Associates, Inc., 2019.

[Piu11] Ian Piumarta. Open, extensible composition models. In *Proceedings of the 1st International Workshop on Free Composition - FREECO '11*, pages 1–5, New York, New York, USA, jul 2011. ACM Press, ACM Press.

[RKBA+13] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation - PLDI '13*, page 519, New York, New York, USA, jun 2013. ACM Press, ACM Press.

[RO10] Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. In *Proceedings of the ninth international conference on Generative programming and component engineering*, pages 127–136, 2010.

[SAB+19] Tetsuya Sato, Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Justin Hsu. Formal verification of higher-order probabilistic programs: reasoning about approximation, convergence, bayesian inference, and optimization. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–30, jan 2019.

[SJSW16] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. On fast large-scale program analysis in datalog. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 196–206, 2016.

[SLTB+06] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 404–415, 2006.

[SS21] Brennan Saeta and Denys Shabalin. Swift for tensorflow: A portable, flexible platform for deep learning. *Proceedings of Machine Learning and Systems*, 3, 2021.

[TB14] Emina Torlak and Rastislav Bodik. A lightweight symbolic virtual machine for solver-aided host languages. *ACM SIGPLAN Notices*, 49(6):530–541, jun 2014.

[Wad90] Philip Wadler. Linear types can change the world! In *Programming concepts and methods*, volume 3, page 5. Citeseer, 1990.

[WNW+21] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. egg: Fast and extensible equality saturation. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–29, jan 2021.

[WP07] Alessandro Warth and Ian Piumarta. Ometa: An object-oriented language for pattern matching. In *Proceedings of the 2007 symposium on Dynamic languages - DLS '07*, page 11, New York, New York, USA, oct 2007. ACM Press, ACM Press.

[YNK+20] Katherine Ye, Wode Ni, Max Krieger, Dor Ma'ayan, Jenna Wise, Jonathan Aldrich, Joshua Sunshine, and Keenan Crane. Penrose: from mathematical notation to beautiful diagrams. *ACM Transactions on Graphics (TOG)*, 39(4):144–1, 2020.