Dataflow Analysis of Hugrs with ascent

Douglas Wilson

June 27, 2024

Outline

1. Datalog
2. An important quantum optimisation problem
3. Conclusion

Example

```
ascent! {
   relation edge(i32, i32);
   relation path(i32, i32);

   path(x, y) <-- edge(x, y);
   path(x, z) <-- edge(x, y), path(y, z);
}
```

- Here is an example that, given the edges of a directed graph, computes whether a path exists between any two nodes.
- A datalog program is a collection of *Horn Clauses*(Horn, Alfred, 1951).
- The most mature open source implementation of Datalog is *souffle*(Scholz, Bernhard and Jordan, Herbert and Subotić, Pavle and Westmann, Till, 2016, Herbert Jordan and Bernhard Scholz and Pavle Subotić, 2016). It works by generating `C++` code from a Datalog program.

Example

```
ascent! {
    relation edge(i32, i32);
    relation path(i32, i32);

    path(x, y) <-- edge(x, y);
    path(x, z) <-- edge(x, y), path(y, z);
}
```

- Think of relations as sets of facts.
- A datalog solver computes all true facts from a set of initial facts.
- A datalog solver mutates relations as it iterates
- This mutation is *monotone*: once a fact exists, it will always exist.

Example

```
ascent! {
   relation edge(i32, i32);
   relation path(i32, i32);

   path(x, y) <-- edge(x, y);
   path(x, z) <-- edge(x, y), path(y, z);
}
```

ascent (Sahebolamri, Arash and Gilray, Thomas and Micinski, Kristopher, 2022) is an implementation of Datalog via `rust` proc-macro.

```
fn main() {
   let mut prog = AscentProgram::default();
   prog.edge = vec![(1, 2), (2, 3)];
   prog.run();
   println!("path: {:?}", prog.path);
}
```

Example

```
ascent! {
   relation edge(i32, i32);
   relation path(i32, i32);

   path(x, y) <-- edge(x, y);
   path(x, z) <-- edge(x, y), path(y, z);
}
```

Datalog is tractable to solve because of *semi-naive evaluation*. Once a fact exists it always exists, therefore during an iteration we know that any new fact we find must depend on a fact that we discovered in the previous iteration.

Conditions and Generative Clauses

```
ascent! {
   relation node(i32, Rc<Vec<i32>>);
   relation edge(i32, i32);

   edge(x, y) <--
      node(x, neighbors),
      for &y in neighbors.iter(),
      if x != y;
}
```

Negation and Aggregation

```
use ascent::aggregators::*;
type Student = u32;
type Course = u32;
type Grade = u16;
ascent! {
   relation student(Student);
   relation course_grade(Student, Course, Grade);
   relation avg_grade(Student, Grade);

   avg_grade(s, avg as Grade) <--
      student(s),
      agg avg = mean(g) in course_grade(s, _, g);
}
```

Lattices

A *Lattice* is a partial order equipped with:

- a binary operation *join*, or least upper bound;
- a binary operation *meet*, or greatest lower bound.

A *Bounded Lattice* has a unique maximum element, called *top* or $\top$ and a unique minimum element called *bottom* or $\bot$.

Lattices In ascent

```
ascent! {
   lattice shortest_path(i32, i32, Dual<u32>);
   relation edge(i32, i32, u32);

   shortest_path(x, y, Dual(*w)) <-- edge(x, y, w);

   shortest_path(x, z, Dual(w + l)) <--
      edge(x, y, w),
      shortest_path(y, z, ?Dual(l));
}
```

- a member of a *lattice* (k, L) is a fact that implies that $(k, l), l \leq L$ is a fact.
- `Dual` is a newtype wrapper that swaps *meet* and *join*. Unfortunately `longest_path` will fail to terminate on any graph with cycles.

Pluggable data structures in ascent

An eqivilence relation:

```
ascent! {
    relation rel(u32, u32);
    rel(a,b) <- rel(b, a)
    rel(a,c) <- rel(a, b), rel(b, c)
}
```

will create N^2 facts.
We can store those facts using only N using a *union-find* data structure:

```
ascent! {
    #[ds(rels_ascent::eqrel)]
    relation rel(u32, u32);
    // ...
}
```

Pluggable data structures in `ascent`

(Sahebolamri, Arash and Barrett, Langston and Moore, Scott and Micinski, Kristopher, 2023) describes an interface to store relations in user-defined data structures.

Users implement several macros, which are then expanded by the `ascent!` macro.

Performance

TODO

Why use a Datalog solver at all?

- Split one hard problem into two slightly easier problems:
  - A Datalog solver
  - A Datalog program

Separate the specification and the implementation of your problem.

Downsides of ascent

- the proc-macro implementation seems to make it difficult to write an extensible tool.

Can't optimise this

```python
@guppy
def circuit(q: Qubit) -> Qubit:
    i = 0
    while i < 2:
        u = h(Qubit())
        if i % 2 == 0:
            q, u = cx(q, u)
        else:
            q, u = cy(q, u)
        i = i + 1
        u.free()
    return q
```

Can optimise this

```python
@guppy
def circuit(q: Qubit) -> Qubit:
  u1, u2 = (Qubit(), Qubit())
  u1 = h(u1)
  q, u1 = cx(q, u1)
  u2 = h(u2)
  q, u2 = cy(q, u2)
  u1.free()
  u2.free()
  return q
```

Dataflow analysis

Dataflow Analysis is a general technique for static program analysis. *SSA* is particularly well suited for this:

- Choose a *Lattice* type with a *bottom*.
- Assign $\perp$ to each edge. (i.e. each "value" in an SSA graph)
- Define a *transfer function* that takes a node and Lattice values for each of its edges, and returns Lattice values for each of its edges.
- Apply the transfer function to each node and mutate the Lattice values for each of its edges by *joining* with the result of the transfer function.
- Iterate the previous step until you reach a fixed point.

Liveness Analysis

Lattice: Define $\perp$ to be *Dead* and $\top$ to be *Live*.
Transfer function: The arguments of `return` are *Live*, the arguments of any node with *Live* results are *Live*.

```
@guppy
def circuit(q: Qubit, theta: float) -> Qubit: # theta is dead
    theta = -theta
    return q # q is live
```

Constant Value Propagation

Define the following lattice:

```
enum ConstantValue { Bottom, Value(u64), Top }
fn join(lhs: ConstantValue, rhs: ConstantValue) -> ConstantValu
  match (lhs, rhs) {
    (Bottom, x) => x,
    (x, Bottom) => x,
    (Value(x), Value(y)) if x == y => Value(x),
    _ => Top
  }
}
```

Transfer Function: this is constant folding.
Consider:

- add(Value(x),Value(y))
- mult(Top,Value(0))

Constant Value Propagation

Define the following lattice:

```
enum ConstantValue { Bottom, Value(u64), Top }
fn join(lhs: ConstantValue, rhs: ConstantValue) -> ConstantValu
  match (lhs, rhs) {
    (Bottom, x) => x,
    (x, Bottom) => x,
    (Value(x), Value(y)) if x == y => Value(x),
    _ => Top
  }
}
```

- After iterating the transfer function, if any node has a $\perp$ input, then that node is *Unreachable*. Perhaps it is in the *else* branch of an always-true *if* statement.
  - Theorem: Values of type *The sum of zero variants* (also called $\perp$). Will always be assigned the lattice value $\perp$.
- Liveness analysis should only mark the inputs of *Reachable* `return` statements.

PartialValue

```
enum PartialValue {
    Bottom,
    Value(hugr::ops::Value),
    PartialSum(HashSet<usize, Vec<PartialValue>>),
    Top
}
```

- *PartialValue* refines the idea of *ConstantValue* to try a little bit harder to not *join* to $\top$.
- *PartialSum* keeps track of which variant it might be, and what those variant's values might be.
- In a *Hugr* all non-function call control flow is controlled by the *tag* of a variant.
  - Conditional
  - TailLoop
  - CFG (arbitrary control flow graph)
- Let's look at `dataflow.rs` in `https://github.com/CQCL/hugr/tree/doug/const-fold2-talk`

Loop unrolling

Imagine a function:

```rust
/// Apply constant value propagation to the inner DFG of a
/// dataflow parent, using `in_values` as the values for the
/// `Input` node. Returns the values for the `Output` node
pub fn cvp_dataflow_parent(hugr: &Hugr, dataflow_parent: Node,
        in_values: Vec[PartialValue]) -> Vec[PartialValue];
```

We can use this to unroll a `TailLoop` node `tl`:

- Do constant value propagation on the `hugr: Hugr`, and retrieve the input values for the `tl`.
  - Call `let out_values = cvp_dataflow_parent(hugr, tl, in_values);`.
  - if `!out_values[0].supports_tag(1)` then the tailloop is proven to iterate at least once.
  - Create a `DFG` before `tl`, containing a copy of `tl`, wired up to the old inputs of `tl`, and with its outputs becoming the new inputs of `tl`.
  - set `in_values = out_values` and iterate.

- Dataflow Analysis is a useful tool and Hugrs are well suited for it to be directly applied. (Heidemann, ????)
- Constant Value Propagation is strong enough to unroll loops in Hugr.
- It is not clear whether `ascent` is an appropriate tool. How can write modular interdependent analases?.