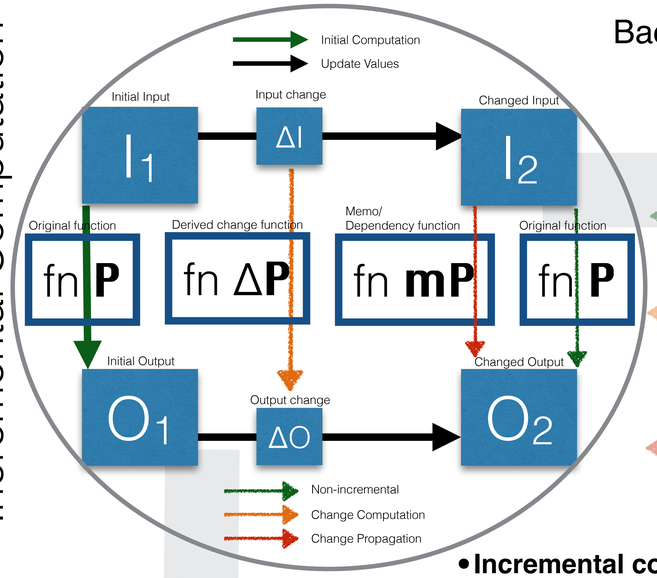
IODyn: A High-Level Language for Incremental Computation

Incremental Computation



Background

Incremental Computation is about updating previously executed computations such that the update is faster than the original time. Sometimes this is not possible, like when most of the data is different. In that case, the best thing to do is re-run the computation. Sometimes, computations are simple enough that we can represent the changes as deltas from the original, and compute directly on the deltas. This type of incremental computation is extremely fast, but does not generalize. For general-purpose incremental computation, we need to keep dynamic dependency graphs that track the flow of changed data throughout a computation.

Problem

- **Incremental computation is critical for efficient, high-performance code.**
- **Caching and reuse require advanced techniques and languages**
- **Current implementations are ad-hoc, and may not be sound.**

IODyn offers a simple language including incremental collections.

Incremental Libraries

Prior incremental code requires specific library calls and knowledge about how to make code incrementally efficient. This means filling code with low-level library calls, even in high-level code.

```
fn max(ml: MemoList) -> Num {
 let l = read(ml);
 let (a,b) = match split(l) {
  None => l.pop(),
  Some((a,b)) => {
   bin_max(
    memo(max(write(a))),
    memo(max(write(b)))
   )
  }
 }
}
```

read/write data for dependency tracking

Explicit subsequence

automatic change propagation

This example shows the calculation of max element of a list by splitting it roughly in half and recursing, running binary max on individual elements. It requires conversions to and from `MemoList`, to provide incremental change notification. It also requires explicit memoization of recursive calls.

Translation Soundness

We are working through a proof that the translation from the IODyn source language to the Typed Adapton target language is sound.

Source translates to target
Source evaluates to terminal

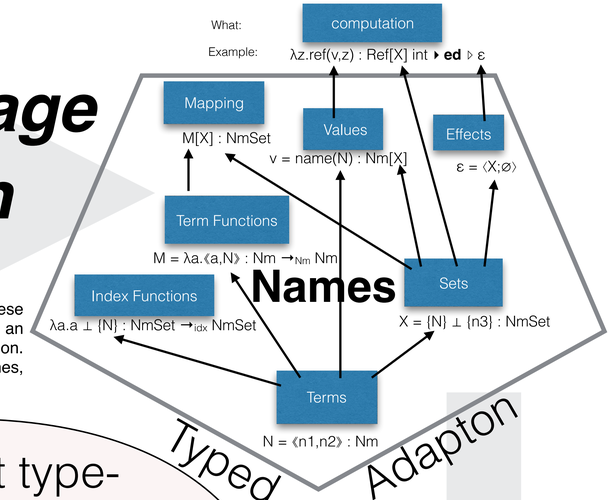
Terminal translates to result
Target evaluates to result

```
if Γ⊢e:C ⇝x Γ⊢e:C▷ε
and σ;e ⇓ σ';t'
…
then
…
Γ'⊢t':C ⇝y Γ'⊢t':C▷⟨∅,∅⟩
and σ;[n/x]e ⇓ σ';t'
```

There are some additional premises, exists variables, and conclusions omitted here for simplicity. They mostly concern store translation and facts about the names (x,y,n above)

Effects

Typed Adapton is a refinement type system with effects. These effects track which names are used in the evaluation on an expression. Here we see the effects of a terminal expression. When evaluated, it does not write to or read from any names, as expected.



Refinement type-checking with Z3

Typed Adapton refinements are typically sets of names. Type-checking judgements dealing with these sets are non-deterministic and difficult to implements. We have been experimenting with the Z3 SMT solver to assist in this task.

```
/// join two effect sets into a new one, accounting for order
pub fn then(&mut self, a:usize,b:usize,c:usize) {
  self.effect_new(c);
  // write sets are disjoint
  writeln!(self.stdin,"(assert (= empty ((_ map and) w{} w{})))",a,b);
  // write before read
  writeln!(self.stdin,"(assert (= empty ((_ map and) r{} w{})))",a,b);
  // c = a union b
  writeln!(self.stdin,"(assert (= w{} ((_ map or) w{} w{})))",c,a,b);
  writeln!(self.stdin,"(assert (= r{} ((_ map or) r{} r{})))",c,a,b);
}
```

There are no well-developed libraries for Z3 in Rust, so we run it as a separate thered and communicate through standard unix pipes. This way, we could learn about Z3 using its native language. This worked well for initial use, but needs to be refined to handle errors better.

This example shows the `then` rule, used for consecutive expression evaluation with let-binding. The second expression must not write(w) to the names read(r) or written by the first expression. The effects of the unified let expression are the union of the effects of its sub expressions.

Background

Adapton handles dependency graphs and has been proven to produce correct results when cache is accessed appropriately. To assist in this access, Typed Adapton uses refinement types to keep track of names representing cache locations. Names act like pointers into cache. Sets of names potentially contain all dynamically allocated cache.

Contributions

The IODyn project will soon be a full pipeline for compiling simple source code into an optimized incremental executable. It will be able to handle changes to large amounts of data, recomputing results in asymptotically less time than initial computation.

IODyn will allow non-experts of IC to use simple designs to build incremental applications that are type-safe, incrementally sound (identical results as from-scratch code), and with performance rivaling more complex programs. IODyn will bring general-purpose incremental computing to a wider range of users.

Bidirectional type checking

Quickhull Sample

```
{fix qh. lam pts. lam line. lam hull.
let complete = { SeqIsEmpty(pts) }
if (complete) then { ret hull } else {
  let mid = { SeqFoldUp(
    pts, (0,0), lam p.ret p, force highest
  ) }
  let hull = {
    let r_line = {{force make_r_line} mid line}
    let r_pts = { [s [sym filt-r]] SeqFilter(
      pts, {force above_line} r_line
    ) }
    [s [sym rec-r]] {force qh} r_pts r_line hull
  }
  let hull = { SeqAppend(hull, mid) }
  let l_line = { {force make_l_line} line mid }
  let l_pts = { [s [sym filt-l]] SeqFilter(
    pts, {force above_line} l_line
  ) }
  [s [sym rec-l]] {force qh} l_pts l_line hull
}} : Seq((nat x nat)) ->
    ((nat x nat) x (nat x nat)) ->
    Seq((nat x nat)) ->
    F Seq((nat x nat))
```

Parametric primitives

Single annotation

IODyn uses a bidirectional type checker, a compromise between full annotations and passing around type variables. Key components may need annotation, but we can type-check higher-order primitives even without needing full-featured parametric polymorphism.

Hints

IODyn uses hints like the ones at the beginning of these lines to indicate sections of code that have certain incremental properties. Here, we're labelling each recursive call, meaning that the path to reach each is not expected to change much. This is the case for quickhull, which deals with points in space. Paths through euclidean space are likely to reach the same general area after small changes.

Macro parsing

Pattern — Conversion

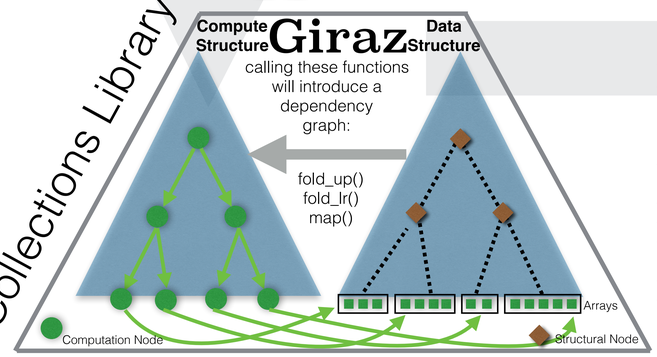
```
macro_rules! make_exp {
…
// thk e
{ thk $($exp:tt)+ } => {{ Exp::Thunk(Rc::new(make_exp![$($exp)+])) }};
// lam r.e (lambda)
{ lam $var:ident . $($body:tt)+ } => {{
  Exp::Lam(stringify![$var].to_string(),Rc::new(make_exp![$($body)+]))
}};
…
}
```

Repeating Pattern — 'Recursive' call

Rust macros aren't able to parse sequences of multiple special characters directly with patterns, so we developed a macro-based fold to do so. We use it for parameter lists, and type signatures that have nested arrows, sum types, and product types.

Rust macros are sophisticated enough to parse simple languages, so we take advantage of this to get a jump start on getting input to our IODyn language.

Collections Library



Background

The Giraz is a data structure for storing and computing with sequences. Internally it uses a tree of incremental data cells, ending in leaves containing arrays of subsequences. The cells provide incrementality, and the arrays provide cache locality for performance. The tree is balanced with a 'canonical form', so that unedited data is never effected by rebalancing.

Functions over the tree like maps and folds are implemented to take advantage of the Adapton engine, so that computations after small edits are tens to hundreds of times faster than the original. Each computation produces a compute structure mirroring the data structure. When data changes, the path up the tree to the root is marked as needing an update. Unmarked nodes are not recomputed.

by Kyle Headley

Project assisted by Matthew Hammer

More info at kyleheadley.github.io

CUPLV
Programming Languages and Verification
University of Colorado Boulder

IODyn Status

December 2017

Planned
In Progress
Complete

Meta Theory
Formal Definition
Implementation