Hilado Deliverable 10.13: A framework for minimal recomputation

Des Small[1] Mark Kettenis[1] Bojan Nikolic[1]

[1] JIVE [2] University of Cambridge

Hilado Meeting, ESO, March 13 2015

Bringing it to the user

Our scenario:

- Imagine incremental development of pipeline
- Change script and rerun
- Products cached and recalculated only when necessary
- Three strands:
  - Compiler theory + Haskell + toy language
  - Casa + execution engine
  - ParselTongue + Swift = ParselSwift
- Uniform approach: single assignment; programs as graphs.

Tiny pipeline with mutation

- Inputs not distinguished from outputs
- Inputs modified in place
- Convenient but opaque

Algorithm 1: Original

```
fn ← "datafile";
data ← read_data(fn, 1);
munge_data(vis=data,
opcode="CAL", p=0.7);
restrain_data(vis=data,
threshold=0.4);
plots ← make_plots(data, b)
```

Algorithm 2: Revised

```
fn ← "datafile";
data ← read_data(fn, 1);
munge_data(vis=data,
opcode="CAL", p=0.7);
restrain_data(vis=data,
threshold=0.5 );
plots ← make_plots(data, b)
```

With return values

- For analysis want outputs and inputs distinguished
- No arguments mutated
- Implies copying, but we *want* copies!
- Partial copies can be cheap (ZFS or AIPS tables)

Algorithm 3: Original

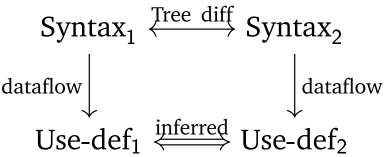
fn ← "datafile";
data ← read_data(fn, 1);
data ← munge_data(vis=data, opcode="CAL", p=0.7);
data ← restrain_data(vis=data, threshold=0.4);
plots ← make_plots(data, b)

Algorithm 4: Revised

fn ← "datafile";
data ← read_data(fn, 1);
data ← munge_data(vis=data, opcode="CAL", p=0.7);
data ← restrain_data(vis=data, threshold=0.5);
plots ← make_plots(data, b)

Syntax tree approach

- Represent syntax as a tree
- Can calculate difference between two trees (Fluri et al)
- Feed into use-def chains (i.e., dependencies)
- Can infer what needs recalculating
- (But we can do better)



Single Static Assignment

- Modern compiler technology!
- "Factorised use-def chains"
- Each variable defined once
- Code reduces to *graph* of definitions

Algorithm 5: Original

fn ← "datafile";
$data_0$ ← read_data(fn, 1);
$data_1$ ← munge_data(vis=$data_0$, opcode="CAL", p=0.7);
$data_2$ ← restrain_data(vis=$data_1$, threshold=0.4);
plots ← make_plots($data_2$, b)

Algorithm 6: Revised

fn ← "datafile";
$data_0$ ← read_data(fn, 1);
$data_1$ ← munge_data(vis=$data_0$, opcode="CAL", p=0.7);
$data_2$ ← restrain_data(vis=$data_1$, threshold=0.5);
plots ← make_plots($data_2$, b)

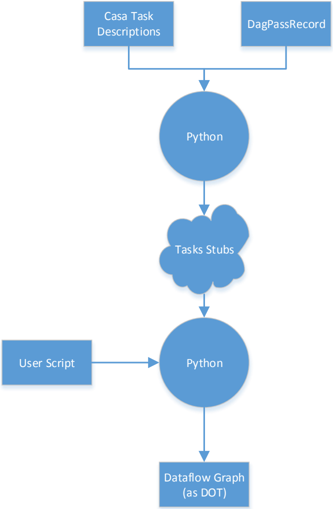
Graph partitioning approach

- Can find equal variables in SSA graph (Alpern et al)
- Can do it across multiple programs
- Can recalculate minimal script with cache
- Running code – it works!

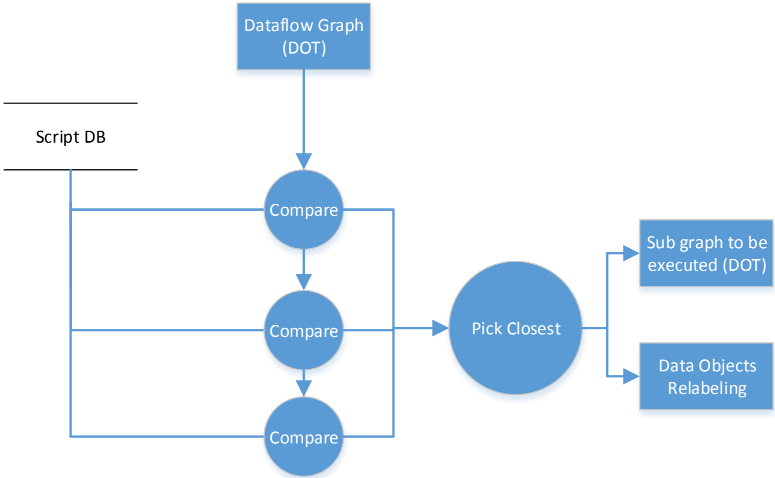
Algorithm 7: Identifying equal variables

```
value_graph = ∅;
for f ∈ program_files do
    ast ← generate_abstract_syntax(f);
    cfg ← generate_control_flow_graph(ast);
    domF ← calculate_dominance_frontiers(cfg);
    ssa ← calculate_SSA_form(cfg, domF);
    valG ← calculate_value_graph(valG);
    value_graph ← value_graph ∪ valG;
end
partition global value graph;
filter out non-variables from partitions
```

From Casa scripts to execution graphs.



Comparing graphs from Casa scripts.



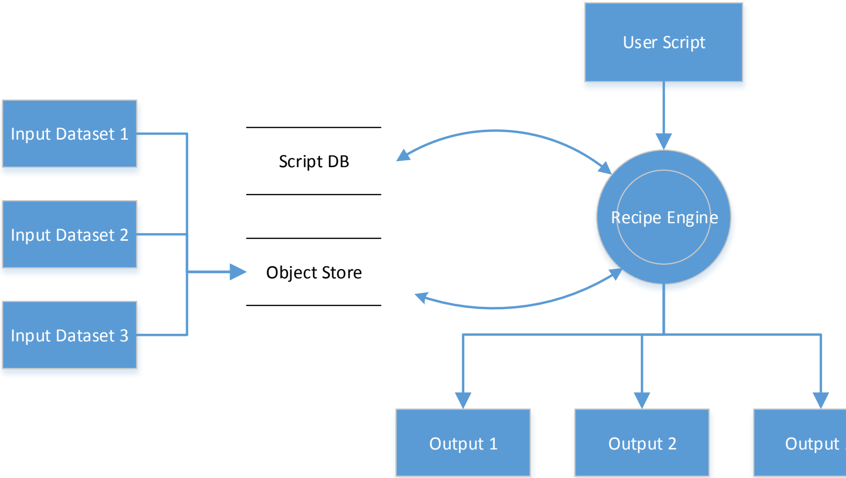


Figure : Execution of Casa scripts with object cache.