Contents

1 Rationale

Many DSLs expose the same core semantics — there is a control flow, possibly with loops, and there is either an imperative destructive assignment or already an equivalent of an SSA or a CPS data flow. Details may differ though, which makes it impossible to reuse a powerful optimisation backend such as LLVM, if there is a significant semantic mismatch between a DSL and LLVM IR.
Yet, there is a lot of optimisation and analysis passes that do not need any low level details. They can operate on a highly abstract IR which only exposes a control flow, an SSA and some abstract "operations" (or "intrinsics"), for a further classification of which (if operation is pure, if it is abelian, etc.) a DSL frontend may provide a feedback.
The abstract transforms that are possible on a generalised IR include, but not limited to:

- Constant folding, given that a concrete DSL frontend is providing evaluators for all of the pure intrinsics.
- Agressive dead code elimination, using the constant folding information. Includes redundant φ loops elimination.
- Control flow simplification, `if`–to–`select` conversion.
- Branch constraint folding
- Transforms based on a loop analysis:
  - Loop invariant motion
  - Loop induction variable detection
  - Strength reduction, if a DSL frontend provides a suitable cost model
  - In-loop and out of a loop constraints (potentially reducing to a constant folding)
  - Iteration dependency analysis, with a help from a DSL frontend model
- Common subexpression elimination and a global value numbering.
- Algebraic transforms, also relying on an algebra model of a DSL frontend.
- Constraint propagation (abstract interpretation).

This may be useful well beyond the scope of low level languages, where SSA registers represent simple values and intrinsics are all simple arithmetic or logic operations. For example, structure composition and decomposition operations (for immutable structures) can also be handled on this abstract level, as well as string operations and many more. Same backend may also be used for an Array-SSA and other higher level forms.
All that is required from a DSL implementation in order to enjoy the SSA-based optimisations is to implement a transformation from this DSL IR into genssa2 and vice versa, and to provide a *model* defining the intrinsics behaviour and contraints. Also, a transformation into an SSA form must be done prior to all the genssa2 based optimisations, using a different abstract IR, genssa.

2 A higher level generic SSA representation

This representation is suitable for doing generic SSA–based transformations, like constant folding, DCE, CSE, partial application, etc.
All the target-specific operations are represented as intrinsic calls. User must provide the intrinsic hooks and annotate explicitly the const calls.
By convention, entry basic block is always called "`entry`".

```
ast genssa2 {
  top = f(globident:nm, type:ret, *argpair:args, code:body);
  code is (.*bblock:bs);
  argpair is (type:t, ident:name);
  bblock =
    b(labident:name, *oppair:ops, term:t);
  oppair is (varident:name, iop:op);
  iop = phi(.*phiarg:args)
      | select(expr:cnd, expr:t, expr:f)
      | call(*attr:a, iident:dst, .*expr:args)
      ;
  switchdst is (expr:v, labident:l);
  term = br(labident:dst)
       | brc(expr:c, labident:tr, labident:fl)
       | switch(expr:v, labident:d, .*switchdst:ns)
       | none()
```

```
     ;
  expr = var(varident:id)
       | glob(globident:id)
       | const(type:t, any:v)
       | other(type:t, any:v)
       ;
  phiarg = a(labident:src, expr:v);
  attr = constcall() | sideeffects() | intrinsic() | external() | other(.*any:as);
  varident is id:v;
  labident is id:v;
}
```

Please note that alloca, load and store are missing. They are supposed to be intrinsics too, if needed at all (e.g., an MBase backend won't need any of those after an SSA–transform).

2.1 Genssa2 utility functions

In order to do the loop analysis, we may need to lower genssa2 to genssa.

```
function depgraph_todot(g) {
   println("digraph X { node [shape=record];");
   iter (f:ts) in g do {
     iter t in ts do
        println(%S<<(t," -> ",f,";"));
   };
   println("}");println("");}

function printgenssa2(g) {
   visit:genssa2(top:g) {
     deep top {
        f ↦ { println(%S<<("FUN: ", nm, " ", ret, "[", args, "]"));
               iter b in body do println(b) }};
     deep oppair: %S<<(name, " = ", op);
     deep bblock {
        b ↦ %S<<(name, ":",
                  foldl( λ(l,r) %S<<(l,"\n   ", r), "", ops),
                  "\n\n   ", t)}}}
```

```
function genssa2_to_genssa(src) {
   egetvar(e) = visit:genssa2(expr:e) {
      once expr { var ↦ id | else ↦ '*none*' }};
   getvars(op) = collector(add, get) {visit:genssa2(iop:op) {
      deep expr { else ↦ add(egetvar(node)) }};
      return get()};

   visit:genssa2(top: src) {
      deep top {
         f ↦ body};
      deep bblock {
         b ↦ { <[t1;t2]> = t(name); 'b'(name, ops⊕t1, t2)}};
      deep oppair: [name;op(name)];
      deep iop(dstreg) {
         phi ↦ 'phi'(dstreg, map(car, args),
                     map a in args do egetvar(cadr(a)))
       | else ↦ 'use'(@getvars(node))};
      deep term(nm) {
         br ↦ [∅; [dst]]
       | brc ↦ [[[nm;'use'(egetvar(c))]];[tr;fl]]
       | switch ↦ [[[nm;'use'(egetvar(v))]];[d;@ns]]
       | none ↦ [∅;∅]};
```

```
    deep switchdst: l;
    deep phiarg { a ↦ [src;v] }}}
```

```
function genssa2_loops(src) {
   gen1 = genssa2_to_genssa(src);
  <[vmap;ngen;DT]> = ssa_transform(gen1, ∅);
   loops = ssa_find_loops(ngen, DT);
   return loops}
```

```
%"Check if intrinsic is pure"
function genssa2_is_pure(env, id) {
   if(id) {
     aif (chk = ohashget(env, id)) {
       return chk('purep')} else ∅}}
```

```
%"Algebraic classification of an intrinsic"
function genssa2_classify(env, id) {
   if (id) {
      aif(chk = ohashget(env, id)) return chk('classify') else ∅}}
```

```
function genssa2_env_getinteger(env, c) {
   aif(chk = ohashget(env, '*get-integer-constant*')) {
      <const(tp, vl)> = c;
       return chk(tp, vl)
   } else ∅}
```

```
%"Check if value is phi or pure"
function genssa2_is_value_pure(env, v)
  visit:genssa2(iop: v) {
    deep iop {
       phi ↦ true
     | select ↦ true
     | call ↦ genssa2_is_pure(env, dst)}}
```

```
%"Cache the definition origin bblocks"
function genssa2_cache_origs(src) {
  oright = mkhash();
  visit:genssa2(top: src) {
      deep bblock {
        b ↦ {ohashput(oright, %Sm<<(name, "__TERM"), name);
              iter o in ops do o(name)}};
      deep oppair:
        λ(bb) ohashput(oright, name, bb)};
  return oright}
```

```
%"Helper function: get exits for a given basic block"
function genssa2_get_exits(bb) collector(add, get) {
 getexits(term) = visit:genssa2 (term: term) {
    deep labident: add(node)};
 visit:genssa2 (bblock: bb) {
    deep term { else ↦ getexits(node) }};
 return get()}
```

```
function genssa2_cache_cfg(src) {
  cfg = mkhash(); bbs = mkhash();
  visit:genssa2(top: src) {
    once bblock {
      b ↦ {ohashput(bbs, name, node);
           ohashput(cfg, name, genssa2_get_exits(node))}}};
  return [cfg;bbs]}
```

3 External language interface

A language environment is a hash table with the following entries:

- `*true?*` — a function that checks if a constant is true
- `*numeric-value*` — a function that returns a numeric value of a constant suitable for a switch index
- Any other entry — an intrinsic function id, holding a dispatch function with the following possible argument values:
  - `constantp` — true if this intrinsic value is constant if its arguments are constant
  - `evalfun` — for the constant intrinsics, an evaluation function taking a list of constant arguments
  - `assocp` — if a binary operation is associative
  - `distribp` — if a binary operation is distributive
  - `isincrement` — if a binary operation is incrementing its first argument by the constant second argument value
  - `isdecrement` — if a binary operation is decrementing its first argument by the constant second argument value
  - `isbounds` — if a binary operation is defining a boundary for its first argument by the constant second argument
  - `iscmp` — if a binary operation is a comparison
  - `changeorderfun` — if available, a function that changes an order of a arguments for a non–associative binary operation (e.g., makes >= from <).
  - `purep` — true if does not have any side effects (memory writes, etc.)

It would make more sense to define a DSL for populating such an environment.

4 Constant folding

Generic constant folding:

- Build a value dependency graph, annotated with the value properties
- Propagate properties, using the following rules
  - A constant function applied to all constant arguments yields a constant value
  - A φ loop with only constant inputs and a constant bound yields a constant value (needs loop analysis data)
  - Side effect instructions are marked so
  - Loops not containing side effects are marked so

4.1 Tagging constant values

```
function genssa2_make_depgraph(src, termp) {
  depgraph = mkhash();

  addedges(f, ts) = {
    x = ohashget(depgraph, f);
    ohashput(depgraph, f, ts⊕x)};

  collect_refs(iop) = collector(add, get) {
```

```
    visit:genssa2(iop:iop) {
      deep expr {
        var ↦ add(id) | else ↦ ∅}};
    return get()};

  collect_term_refs(term) = collector(add, get) {
    visit:genssa2(term:term)  {
      deep expr {
        var ↦ add(id) | else ↦ ∅}};
    return get()};

  visit:genssa2 (top: src) {
    deep bblock {
      b ↦ if(termp) {
            refs = collect_term_refs(t);
            addedges(%Sm<<(name, "__TERM"), refs)}};
    deep oppair: op(name);
    deep iop(dst) {
      else ↦ {
        refs = collect_refs(node);
        addedges(dst, refs);
        return node}}};

  depgraph1 = mkhash();
  hashiter(λ (f, ts) {
      ohashput(depgraph1, f, unifiq(ts))
    }, depgraph);

  return depgraph1}
```

```
function genssa2_cache_defs(src) {
  ht = mkhash();
  visit:genssa2(top:src) {
    deep oppair: ohashput(ht, name, op)
  };
  return ht}
```

```
function genssa2_is_const(env, ctab, op) {
   getdst(id) = {
      aif (chk = ohashget(env, id))
        return chk('constantp')
        else ∅};
   checkarg(v) =
     visit:genssa2(expr: v) {
       deep expr {
         const ↦ true
       | var ↦ ohashget(ctab, id)
       | else ↦ ∅}};
   visit:genssa2(iop: op) {
    deep attr {
      constcall ↦ true
    | else ↦ ∅};
    deep iop {
      phi ↦ ∅ // Must be reduced elswhere
    | select ↦ ∅
    | call ↦ {
        d = getdst(dst);
        const_a = foldl(λ(x, y) x||y, ∅, a);
        if (d || const_a)
          foldl(λ(x,y) x&&y, true,
                map(checkarg, args))
```

```
      else ∅}}}}
```

```
function genssa2_tag_constants(env, src, depgraph) {
   defs = genssa2_cache_defs(src);
   ctab = mkhash();

   // Initial pass
   visit:genssa2(top:src) {
      deep oppair: {
         if (genssa2_is_const(env, ctab, op))
            ohashput(ctab, name, true)}};

   // Invert the dependency graph
   idep = mkhash();
   hashiter(λ(k,vs) {
         iter v in vs do
           ohashput(idep, v, unifiq(k:ohashget(idep, v)))}, depgraph);

   // Follow the inverse dependency graph
   seed = hashmap(λ(k,v) k, ctab);
   vis = mkhash();
   iter s in seed do ohashput(vis, s, s);
   nextfront = unifiq(map append n in seed do ohashget(idep, n));
   do loop(front = nextfront) {
      nxt = map append n in front do {
         vl = ohashget(defs, n);
         if (genssa2_is_const(env, ctab, vl)) {
           ohashput(vis, n, n);
           return ohashget(idep, n)
         } else ∅};
      if(nxt) loop(nxt)};
   return defs:ctab}
```

4.2 Evaluating constants

```
function genssa2_eval_constant(env, defs, dst, cache, op) {
  getdst(id) = {
              aif (chk = ohashget(env, id)) {
                 ret = chk('evalfun');
                 if(ret) ret else λ(args) ∅
              } else λ(args) ∅};
  evalarg(e) =
      visit:genssa2(expr: e) {
         deep expr {
            const ↦ node
         | var ↦ {
              v = ohashget(defs, id);
              genssa2_eval_constant(env, defs, id, cache, v)}
         | else ↦ ccerror('IMPOSSIBLE'())}};
  aif (chk = ohashget(cache, dst)) return chk
  else {
    tmp = visit:genssa2(iop:op) {
       deep iop {
          phi ↦ ccerror('NOT-HERE-PLEASE'())
        | select ↦ ccerror('NOT-HERE-PLEASE'())
        | call ↦ {
            ef = getdst(dst);
            if (not(ef)) ccerror('NO-EVAL-FUNCTION'());
            ret = ef(map a in args do evalarg(a));
            if (ret) ret else ∅}}};
```

```
    ohashput(cache, dst, tmp);
    return tmp}}
```

If a conditional branch is dependent on a constant, it can be replaced with an unconditional branch, and if any parts of a CFG become unreachable, they may be safely eliminated, potentially making φ nodes constant.

```
function genssa2_is_const_value(env, defs, ctab, vl) {
  visit:genssa2(expr: vl) {
   once expr {
     var ↦ {
        aif (chk = ohashget(defs, id)) genssa2_is_const(env, ctab, chk)
        else ∅}
   | const ↦ true
   | else ↦ ∅}}}
```

```
function genssa2_eval_constant_value(env, defs, dst, cache, vl) {
 visit:genssa2(expr: vl) {
   once expr {
     var ↦ {
        aif (chk = ohashget(defs, id)) {
           aif (ret = genssa2_eval_constant(env, defs, dst, cache, chk))
               ret
           else node}
        else node}
   | else ↦ node}}}
```

```
%"Simulate the switch instruction behaviour for
  a given number, return a taken branch and a list
  of dropped branches"
function genssa2_eval_switch(env, vl, ns, d) {
  // TODO!
  ccerror('TODO'());
  return d:∅
  }
```

```
%"Do one pass of constant flow unrolling"
function genssa2_unroll_constant_flow_step(env, defs, ctab, src, story) {
  evalcache = mkhash();
  istrue = ohashget(env, '*true?*');
  getnumeric = ohashget(env, '*numeric-value*');
  visit:genssa2(top:src) {
     deep bblock {
        b ↦ t(name)};
     deep term(bbname) {
        br  ↦ ∅
     |  brc ↦
           if (genssa2_is_const_value(env, defs, ctab, c)) {
              tname = %Sm<<(bbname, '-term');
              vl = genssa2_eval_constant_value(env, defs,
                     tname, evalcache, c);
              if (istrue(vl)) {
                story('constant_branch'(bbname, tr));
                if (not(tr===fl))
                    story('eliminate_branch'(bbname, fl));
              } else {
                story('constant_branch'(bbname, fl));
                if (not(tr===fl))
                   story('eliminate_branch'(bbname, tr))}}
     |  switch ↦
```

```
      if (genssa2_is_const_value(env, defs, ctab, v)) {
        tname = %Sm<<(bbname, '-term');
        vl = genssa2_eval_constant_value(env, defs, tname,
                  evalcache, v);
       <taken:dropped> = genssa2_eval_switch(env, vl, ns, d);
        story('constant_branch'(bbname, taken));
        iter d in dropped do
          story('eliminate_branch'(bbname, d))}}}}
```

```
%"Execute the eliminate_branch and constant_branch suggestions"
function genssa2_rewrite_cfg(src, commands) {
  constbrs = mkhash(); drops = mkhash();
  {iter c in commands do
     match c with
       'constant_branch'(f, t) ↦
          ohashput(constbrs, f, t)
     | 'eliminate_branch'(f, t) ↦
          hashput(drops, %S<<(f, "->", t), true)};

  isconstbranch(bb) = ohashget(constbrs, bb);
  dropedge(src, dst) = hashget(drops, %S<<(src, "->", dst));

  visit:genssa2(top: src) {
    deep bblock {
      b ↦ mk:node(ops = map o in ops do o(name), t = t(name))};
    deep oppair: λ(bb) [name; op(bb)];
    deep iop(bb) {
      // Phis that drop all or all but one of their arguments
      // must be handled in another pass
      phi ↦ mk:node(args = map append a in args do a(bb))
    | else ↦ node};
    deep term(bb) {
      else ↦ {
        aif(chk = isconstbranch(bb))
          'br'(chk)
        else node}};
    deep phiarg(bb) {
      a ↦ {
        if (dropedge(src, bb)) ∅
        else [node]}}}}
```

```
%"Fold the foldable constants"
function genssa2_fold_constants(env, src, defs, ctab, chgp) {
  evalcache = mkhash();
  visit:genssa2(top: src) {
    deep expr {
      var ↦ if (genssa2_is_const_value(env, defs, ctab, node)) {
               ret = genssa2_eval_constant_value(env, defs,
                        id, evalcache, node);
               if (ret) {
                 chgp := true;
                 return ret
               } else node
             } else node
    | else ↦ node}}}
```

4.3 Handling calls with select arguments

There is a special case where calls are not entirely constant, but have a mixture of constant and select arguments, where select, in turn, have one of its branches constant. Such a call can fused inside a select instead, which may or may not lead to further optimisations.

Ideally, this must be a backtracking point — something to consider for the future.

```
function genssa2_rewrite_constant_selects(env, defs, ctab, src, chgp) {
  evalcache = mkhash();
  istrue = ohashget(env, '*true?*');
  rewht = mkhash();
  rewrite(id) =
     aif(chk = ohashget(rewht, id)) { chgp := true; chk}
                          else { 'var'(id)};
  visit:genssa2(top: src) {
     deep oppair: op(name);
     deep iop(dst) {
        select ↦
         if (genssa2_is_const_value(env, defs, ctab, cnd)) {
           vl = genssa2_eval_constant_value(env, defs, dst, evalcache, cnd);
           nv = if (istrue(vl)) t else f;
           ohashput(rewht, dst, nv)}
       | else ↦ ∅}};
  visit:genssa2(top: src) {
    deep expr {
      var ↦ rewrite(id)
    | else ↦ node}}}
```

```
function genssa2_fold_selects_rewrite(defs, src, rewrites) {
   // 1. Cache the rewrite commands
   ht = mkhash();
   iter rewrite(bb, dstreg, a, fn, args, cnd) in rewrites do
      ohashput(ht, dstreg, [bb;a;fn;args;cnd]);
   // 2. Apply the rewrites
   doselect(br, op, srcvl) =
      visit:genssa2(iop: op) {
        deep iop {
           select ↦ if (br) t else f
         | else ↦ srcvl}};
   makeargs(br, args) = {
      map a in args do {
         visit:genssa2(expr: a) {
          deep expr {
            var ↦ {
               aif (chk = ohashget(defs, id))
                     doselect(br, chk, node)
                else node}
          | else ↦ node}}}};
   visit:genssa2(top: src) {
      deep bblock {
         b ↦ {
                nops = map append o in ops do o(name);
                mk:node(ops = nops)}}};
      deep oppair: λ(bb) collector(add, get) {
         nop = op(add, bb, name);
         nxt = get();
         nxt ⊕ [mk:node(op = nop)]};
      deep iop(add, bb, dstreg) {
         call ↦ {
            match ohashget(ht, dstreg) with
               [bb1;a1;fn1;args1;cnd] ↦ symbols(new_t, new_f) {
                  add([new_t; 'call'(a1, fn1, @makeargs(true, args1))]);
                  add([new_f; 'call'(a1, fn1, @makeargs(∅, args1))]);
                  return 'select'(cnd, 'var'(new_t), 'var'(new_f));
               }
```

```
          | else ↦ node}
      | else ↦ node}}}
```

```
%"Handle the case of a pure call of similar select arguments,
 if one set of select branches is constant"
function genssa2_fold_selects(env, defs, ctab, src, chgp) collector(add, get) {
 get_select_shape(a, shp) = {
    checkvar(id) = {
       aif (chk = ohashget(defs, id))
         visit:genssa2(iop:chk) {
           deep iop {
              select ↦ {
                nshp =[cnd; genssa2_is_const_value(env, defs, ctab, t);
                            genssa2_is_const_value(env, defs, ctab, f)];
                if (shp) {
                   if (iso(shp, nshp)) shp
                   else ∅
                } else nshp}
           | else ↦ ∅}}};
    visit:genssa2(expr:a) {
       deep expr {
         var ↦ checkvar(id)
       | const ↦ shp
       | else ↦ ∅}}};
 uniform_select_args(args) =
   do loop(as = args, shp = ∅) {
      match as with
        a:tl ↦ {
          aif (chk = get_select_shape(a, shp))
                loop(tl, chk)
           else ∅}
      | else ↦ shp};
 visit:genssa2(top:src) {
    deep bblock { b ↦ iter o in ops do o(name) };
    deep oppair: λ(bb) op(bb, name);
    deep iop(bb, dstreg) {
       call ↦
         // TODO: may want to use some cost model here
         if (genssa2_is_pure(env, dst)) {
           chk = uniform_select_args(args);
           if (chk && {<[x;y;z]> = chk; y||z}) {
              add('rewrite'(bb, dstreg, a, dst, args, car(chk)))}}
     | else ↦ ∅}};
 rewrites = get();
 if (rewrites) {
      chgp := true;
      return genssa2_fold_selects_rewrite(defs, src, rewrites)}
 else return src
}
```

5 Dead code elimination steps

DCE can be combined with a constant folding iteratively. If a branch condition is becoming a constant, a conditional branch can be replaced with an unconditional one, potentially leaving dangling CFG nodes.
Removing such basic blocks may also simplify remaining φ nodes, either fully eliminating them (leaving only one entry edge) or opening them for further optimisations (select transformation, etc.).

```
%"Find reachable basic blocks"
function genssa2_reachable_bbs(src) {
  live = mkhash(); bbhash = mkhash();
```

```
  visit:genssa2 (top: src) {
    deep bblock {
      b ↦ ohashput(bbhash, name, node)}};
  do loop(front = ['entry']) {
    nexts = map append bb in front do {
      if(not(ohashget(live, bb))) {
        ohashput(live, bb, bb);
        return genssa2_get_exits(ohashget(bbhash, bb))}};
    if(nexts) loop(unifiq(nexts))};
  return live}
```

```
%"Remove all the basic blocks not in the live list"
function genssa2_remove_dead_bbs(src, live, chgref)
 visit:genssa2(top: src) {
   deep code: { map append b in bs do b };
   deep bblock {
     b ↦ if (ohashget(live, name)) [node] else {chgref := true; ∅}}}
```

```
%"Get number of uses for each register.
 Cyclic phi references are handled elsewhere."
function genssa2_count_refs(src) {
   cnt = mkhash();
   inccnt(nm) = {
     aif (chk = ohashget(cnt, nm)) ohashput(cnt, nm, chk+1)
        else ohashput(cnt, nm, 1)};
   visit:genssa2(top: src) {
     deep expr {
        var ↦ inccnt(id)
      | else ↦ ∅}};
   return cnt}
```

```
%"Remove the unused instructions if they have no side effects"
function genssa2_remove_dead_regs(env, src, reghash, chgref) {
   isdead(reg) = {
      not(ohashget(reghash, reg));
   };
   kill() = {chgref := true};

   visit:genssa2(top: src) {
      deep bblock {
         b ↦ mk:node( ops = map append op in ops do op )};
      deep oppair: {
        <k:v> = op(name);
         if(not(k)) {kill(); ∅} else [mk:node(op = v)]};
      deep iop(dstreg) {
         phi ↦ if (isdead(dstreg)) ∅:node else true:node
       | select ↦ if (isdead(dstreg)) ∅:node else true:node
       | call ↦ if (isdead(dstreg) && genssa2_is_pure(env, dst)) ∅:node
                 else true:node}}}
```

```
%"Perform a single DCE step: remove unreferenced BBs,
 remove non-side-effect calls with zero uses"
function genssa2_dce_step(env, src, chgref) {
  live = genssa2_reachable_bbs(src);
  src1 = genssa2_remove_dead_bbs(src, live, chgref);
```

```
  reghash = genssa2_count_refs(src1);
  return genssa2_remove_dead_regs(env, src1, reghash, chgref)}
```

```
%"Eliminate the redundant phi nodes"
function genssa2_cull_phis(src, chgp) {
   rewrites = mkhash();
   rewrite(id0) =
     do loop(v = id0) {
        aif (chk = ohashget(rewrites, v)) {
          chgp := true;
          visit:genssa2(expr:chk) {
            once expr{
               var ↦ loop(id)
             | else ↦ node}}}
        else 'var'(v)};

   getphiarg(a) = visit:genssa2(phiarg:a) {
     once phiarg { a ↦ v }};

   checkphi(dst, args) = {
     match args with
        [one] ↦ {ohashput(rewrites, dst, getphiarg(one)); true}
      | else ↦ ∅};

   nxt=visit:genssa2(top: src) {
     deep bblock {
       b ↦ mk:node(ops = map append o in ops do o)};
     deep oppair: {
       v = op(name);
       if(v) [mk:node(op = v)] else ∅};
     deep iop(dst) {
       phi ↦ if(checkphi(dst, args)) ∅ else node
     | else ↦ node}};

   return visit:genssa2(top:nxt) {
     deep expr {
       var ↦ rewrite(id)
     | else ↦ node}}}
```

5.1 Inner loop

This is an inner loop inside of the outer optimisation loop. It makes sense to short DCE with constant folding in order to avoid doing more costly passes (loop analysis, etc.) too often for any resulting changes to propagate.

```
%"Interleave folding and DCE steps
  until fixed point is reached."
function genssa2_fold_and_kill(env, src0, chg_g) {
   do loop(src = src0) {
      chgp = mkref(∅);
      dg = genssa2_make_depgraph(src, ∅);
      <defs:ctab> = genssa2_tag_constants(env, src, dg);

      fsrc = genssa2_fold_constants(env, src, defs, ctab, chgp);
      ssrc = genssa2_rewrite_constant_selects(env, defs, ctab, fsrc, chgp);
      fssrc = genssa2_fold_selects(env, defs, ctab, ssrc, chgp);
      collector(storyadd, storyget) {
         genssa2_unroll_constant_flow_step(env, defs, ctab, fssrc, storyadd);
         cmds = storyget();
         if (cmds) chgp := true;
         src1 = genssa2_rewrite_cfg(fssrc, cmds);
         src2 = genssa2_cull_phis(src1, chgp);
```

```
    next = genssa2_dce_step(env, src2, chgp);
    if (^chgp) { chg_g := true; loop(next) } else next}}}
```

5.2 Those pesky circular φ dependencies

Reducing constants and eliminating branches may lead to dead φ nodes that are not used anywhere else but in their own definitions. Such cycles can be rather complex and involve multiple φ nodes and multiple pure intrinsics. We're going to mark such chains using the following (naive and suboptimal) algorithm:

- Find loops in register dependencies:
  - Node belongs to a loop iff there is a mutual dependency on all other nodes in this loop (i.e., loop is a strong–connected sub–graph, so we can use Tarjan algorithm to find them all).
- Count the uses of each of the identified cyclic sub–graphs that do not belong to them
- Eliminate those that do not have any external uses and only contain pure nodes

```
%"Use Tarjan algorithm to identify dependency loops"
function genssa2_dependency_loops(dgraph) {
  igraph = graph2graph(dgraph);

  clusters = mkhash(); revht = mkhash();
  addcluster(c) = { // Make sure each cluster is added only once
    next = unifiq(map append i in c do {
                  x = ohashget(revht, i);
                  if (not(x)) ['new']
                    else x});
    newp = filter next as next==='new';
    if (newp || (length(next)>1)) {
      nm = gensym();
      ohashput(clusters, nm, c);
      iter i in c do ohashput(revht, i, nm:ohashget(revht, i))}};
  getclusters() = hashmap(λ(k,v) v, clusters);

  iter i in unifiq(map append d in dgraph do d) do {
    if (not(ohashget(revht, i))) {
     clusters = tarjan(igraph, i);
     iter c in clusters do
       match c with
         [one] ↦ ∅
        | else ↦ addcluster(c)}};

  return getclusters()}
```

```
function genssa2_dce_phi(env, src, chgref) {
  // 1. Make the preparations
  igraph = genssa2_make_depgraph(src, 'terms');
  dgraph = hashmap(λ(k,v) k:v, igraph);

  revgraph = mkhash();
  iter (f:ts) in dgraph do iter t in ts do {
    ohashput(revgraph, t, unifiq(f:ohashget(revgraph, t)))};

  //TODO: kill this debugging output
  // depgraph_todot(dgraph);
  ///////////////////////////////
  loops = genssa2_dependency_loops(dgraph);
  dcache = genssa2_cache_defs(src);

  // 2. Leave only the pure loops
  pureloops = filter cs in loops as
```

```
                    foldl(λ(l,r) l&&r, true,
                       map c in cs do
                           genssa2_is_value_pure(env, ohashget(dcache, c)));

   // 3. Find the external uses for each loop
   markloops = map cs in pureloops do {
                    refs0 = map append c in cs do
                                 ohashget(revgraph, c);
                    refs = unifiq(refs0);
                    extrefs = filter r in refs as not(memq(r, cs));
                    return [cs; extrefs]};
   // 4. Iteratively eliminate loops with no external uses, removing the
   //    eliminated loop contents from the other loops use lists.
   dead = mkhash();
   do loop(m = markloops) {
      chg = mkref(∅);
      next = map append [cs;rfs] in m do {
         rfs1 = filter r in rfs as not(ohashget(dead, r));
         if(not(rfs1)) {
            iter c in cs do ohashput(dead, c, c);
            chg := true;
            return ∅
         }  else return [[cs; rfs1]]};
      if(^chg) loop(next)};
   tokill = hashmap(λ(k,v) k, dead);
   if (tokill) chgref := true;
   // 5. Execute the kill list
   tokillh = mkhash();
   iter k in tokill do ohashput(tokillh, k, k);
   visit:genssa2(top: src) {
      deep bblock {
         b ↦ mk:node( ops = map append op in ops do op )};
      deep oppair: {
          if (ohashget(tokillh, name)) ∅ else [node]}}}
```

6 φ to select

Select can be easier to reason about than φ.
We will temporarily convert φ nodes to select if it is possible to infer a single simple logical expression that defines all the φ paths. We'll use the following rules:

- φ node in basic block C have only two entries, from A and B.
- There are only two paths (excluding reducible flow and escape routes) from D to C — one via A and another via B, where D is the nearest common dominator for A, B and C. This rule also means that the only possible common nodes on both paths are D and C.
- φ entries only refer to the values visible in D (i.e., defined in the basic blocks that dominate D).

In this case we can rewrite eligible φ nodes in C as *select*$(D_c, \ldots)$.

```
%"Find a nearest common dominator for [lst],
  return basic block id or []"
function genssa2_find_nearest_dominator(domtree, lst) {
   // 1. Find all the common dominators
   ls = map k in lst do ohashget(domtree, k);
   common = foldl(λ(l, r)
                     {filter r as memq(r, l)},
                  car(ls), cdr(ls));
   // 2. Eliminate those that dominate anything else in this list
   inv = mkhash();
   hashiter(λ (k, vs) {
        iter v in vs do
           if(not(v===k)) ohashput(inv, v, k:ohashget(inv, v))
```

```
    }, domtree);
  common2 = filter c in common as {
      tmp = ohashget(inv, c);
      not(foldl(λ(l,r) (l||r), ∅, map t in tmp do memq(t, common)))};
  match common2 with
    [one] ↦ one
  | else ↦ ∅}
```

```
%"Return a path from f to t, excluding escape routs"
function genssa2_mark_path(domtree, cfg, f, t, c) {
  // Leave only the relevant part of a CFG:
  //    - Only the nodes dominated by f
  //    - Only the nodes from which t is reachable
  //      (taint the CFG backwards)

  // 1. Inverted CFG, inverted domtree
  inv = mkhash();
  hashiter(λ(k,vs)
     { iter v in vs do ohashput(inv, v, k:ohashget(inv, v))}, cfg);

  revdomtree = mkhash();
  hashiter(
    λ(k, vs)
     { iter v in vs do
        ohashput(revdomtree, v,
                 unifiq(k:ohashget(revdomtree, v)))},
    domtree);

  // 2. Taint inverted CFG
  vis = mkhash(); ohashput(vis, c, c);
  do loop(front = [t]) {
     next = map append ff in front do
              {
                nxt = ohashget(inv, ff);
                ohashput(vis, ff, ff);
                map append n in nxt do
                  if (ohashget(vis, n)) ∅ else [n]};
     if (next) loop(next)};

  // 3. Leave only the dominated
  dl = ohashget(revdomtree, f); dom = mkhash();
  iter d in dl do ohashput(dom, d, d);
  collector(add, get) {
     hashiter(λ(k, v) if(ohashget(dom, k)) add(k), vis);
     return get()}}
```

```
%"Check if there are two paths that do not overlap,
  give a proper a-b order wrt. d exits order"
function genssa2_find_paths(domtree, cfg, d, a, b, c) {
   // 1. Get paths from D to A and D to B
   p1 = genssa2_mark_path(domtree, cfg, d, a, c);
   p2 = genssa2_mark_path(domtree, cfg, d, b, c);
   // 2. Check if the paths have nothing in common besides D
   m = map append x in p1 do if(x===d) ∅
                     else if(x===c) ∅
                     else if(memq(x,p2)) [x] else ∅;
   if (m) return ∅
   else {
     // 3. Find which edge of D leads to A (there are only two, so the
     //    other one goes to B)
     dexits = ohashget(cfg, d);
```

```
    a0 = car(dexits); // we'll look for this one
    if (memq(a, p1)) return [true;p1;p2]
                     else return [∅;p1;p2]
  }}
```

```
%"Get a branching condition for a given basic block"
function genssa2_get_condition(bb)
  visit:genssa2(bblock: bb) {
    deep bblock { b ↦ t };
    deep term {
       brc ↦ c
     | else ↦ ∅}}
```

```
%"Check if there is a potential phi to select pattern"
function genssa2_detect_select_pattern(
            env, domtree, cfg, defs, bbs, a, b, c)
{
  d = genssa2_find_nearest_dominator(domtree, [a;b;c]);
  if (d) {
    pths = genssa2_find_paths(domtree, cfg, d, a, b, c);
    match pths with
      [de;pa; pb] ↦ {
        aif(cnd = genssa2_get_condition(ohashget(bbs, d)))
         return [d; de; cnd]
        else ∅}
    | else ↦ ∅ // disqualified
  }
}
```

```
%"Check if code motion is possible"
function genssa2_attempt_motion(
            env, d, domtree, defs, defsh, src, varid,
            dset, moveadd, vh)
{  op = ohashget(defsh, varid);
  chk = visit:genssa2(iop: op) {
    deep expr {
      var ↦
       if (not(ohashget(vh, id))) {
         vsrc = ohashget(defs, id);
         ohashput(vh, id, id);
         ret = if (not(vsrc)) 'ok'
               else if (memq(vsrc, dset)) 'ok'
                 else genssa2_attempt_motion(env, d, domtree,
                        defs, defsh, src, id, dset, moveadd, vh);
         ohashput(vh, %Sm<<(id, "-val"), ret);
         ret}
       else ohashget(vh, %Sm<<(id, "-val"))
    | else ↦ node};
    deep iop {
      phi ↦ ∅ // immediate disqualification
    | select ↦ cnd&&t&&f
    | call ↦ {
        if (genssa2_is_pure(env, dst)
            && foldl(λ(l,r) l&&r, true, args)) 'ok'
        else ∅}
    | else ↦ ∅}};
  if (chk) {
```

```
    moveadd('move'(varid, d)); return 'ok'
  } else ∅}
```

```
%"Try to apply suspected phi to select rewrites"
function genssa2_try_phi_rewrites(env, domtree, src, rewrites, chgp)
collector(moveadd, moveget) {
  // 1. Cache the rewrite commands
  rs = mkhash();
  iter try_to_rewrite(bb, d, neg, cnd) in rewrites do {
    ohashput(rs, bb, [d;neg;cnd]);
  };
  // 2. Cache the definition origins
  defs = mkhash(); defsh = mkhash();
  visit:genssa2(top: src) {
    deep bblock { b ↦ iter o in ops do o(name) };
    deep oppair:
      λ(bb) {
        ohashput(defs, name, bb);
        ohashput(defsh, name, op)}};
  // 3. For each bb to be rewritten, for each phi node,
  //    check if source variables are defined in
  //    the basic blocks that dominate D

  getarg(a) = visit:genssa2(phiarg:a) {
    deep phiarg { a ↦ v }};

  checkarg(d, dset, a, rwadd) = visit:genssa2(phiarg:a) {
    deep phiarg { a ↦ v };
    deep expr {
      var ↦ {
        src = ohashget(defs, id);
        // If the origin is not visible in D,
        // we may still consider moving it there
        if (not(src)) true else
        if (memq(src, dset)) true
          else genssa2_attempt_motion(env, d, domtree, defs, defsh,
                      src, id, dset, rwadd, mkhash())}
    | else ↦ true}};

  pass1 =  visit:genssa2(top: src) {
    deep bblock { b ↦ mk:node(ops = map o in ops do o(name))};
    deep oppair: λ(bb) mk:node(op=op(bb, name));
    deep iop(bb, dst) {
       phi ↦
         {match ohashget(rs, bb) with
            [d;nneg;cnd] ↦ collector(rwadd, rwget) {
              <[a1;a2]> = args;
              dset = ohashget(domtree, d);
              if (checkarg(d, dset, a1, rwadd) &&
                  checkarg(d, dset, a2, rwadd)) {
                // Confirmed, we can rewrite it.
                chgp := true;
                iter i in rwget() do moveadd(i);
                if (nneg)
                  return 'select'(cnd, getarg(a1), getarg(a2))
                else
                  return 'select'(cnd, getarg(a2), getarg(a1))
              } else node}
          | else ↦ node}
    |  else ↦ node
    }};
  moves = moveget();
```

```
if (moves) { // do the code motion pass
  mh = mkhash(); revmh = mkhash();
  iter move(id, dst) in moves do {
       ohashput(mh, id, dst);
       ohashput(revmh, dst,
                 ohashget(revmh, dst)⊕
                    [[id;ohashget(defsh, id)]])};
  pass2 =
   visit:genssa2(top: src) {
     deep bblock {
        b ↦ {
           mv = ohashget(revmh, name);
           mk:node(ops = (map append o in ops do o)⊕mv)}};
     deep oppair: if (ohashget(mh, name)) ∅ else [node]};
  return pass2
} else return pass1}
```

```
%"An interface function, detect and apply phi to select rewrites"
function genssa2_detect_selects(env, src, chgp) {
   // 0. Build a CFG, cache defs and basic blocks
   cfg = mkhash(); bbs = mkhash();
   visit:genssa2(top: src) {
      once bblock {
         b ↦ {ohashput(bbs, name, node);
               ohashput(cfg, name, genssa2_get_exits(node))}}};

   defs = genssa2_cache_defs(src);

   // 1. Buid domtree
   domtree = graph_dominators(cfg, 'entry');

   // 2. For each potentially perspective phi node,
   //    check for select pattern presence
   candidates = mkhash();
   visit:genssa2(top: src) {
      deep bblock {
         b ↦ iter o in ops do o(name)};
      deep oppair: λ(bb) op(bb);
      deep phiarg { a ↦ src };
      deep iop(bb) {
         phi ↦
            if(length(args)==2)
              if (not(ohashget(candidates, bb))) {
                 ohashput(candidates, bb, args)}
       | else ↦ ∅}};
   clist = hashmap(λ(k,v) k:v, candidates);

   collector(add, get) {
     iter [c;a;b] in clist do {
        chk = genssa2_detect_select_pattern(env, domtree, cfg, defs,
                                             bbs, a, b, c);
        if (chk) {
          <[d;neg;cnd]> = chk;
           add('try_to_rewrite'(c, d, neg, cnd))
        }};
     rewrites = get();
     if (rewrites)
         return genssa2_try_phi_rewrites(env, domtree, src, rewrites, chgp)
     else return src}}
```

7 Clean up CFG after all the constant folding and ADCE

Constant folding combined with a φ to select conversion may leave lots of needless branches.
Basic blocks are merged if there is an unconditional branch $A \to B$ and B has only one predecessor. Conditional branches are eliminated if both paths are empty, leading to the same basic block and there are no φ nodes in it.

```
function genssa2_kill_dangling(src) {
  exits = mkhash();
  ohashput(exits, 'entry', 'entry');
  addexits(t) = visit:genssa2(term: t) { deep labident: ohashput(exits, node, node) };
  visit:genssa2(top: src) { once term { else ↦ addexits(node) }};
  visit:genssa2(top: src) {
   deep top {
     f ↦ mk:node(body = map append b in body do b)};
   deep phiarg {
     a ↦ if(ohashget(exits, src)) [node] else ∅};
   deep iop { phi ↦ mk:node(args = map append a in args do a)
            | else ↦ node};
   deep bblock {
     b ↦ if(ohashget(exits, name)) [node] else ∅}}}
```

```
function genssa2_paths_converge(cfg, rev, bbs, f) {
  // Back propagate from f to check if paths leading to it
  // can be eliminated.
  //
  // This is only done for the nodes with multiple preds and no
  // phis.
  //
  // Each pred. node on each edge is only followed iff it is empty
  // and have only one exit.
  vis = mkhash();
  ret = do loop(front = [f], done = ∅) collector(add, get) {
     nxs = map append f in front do if (not(ohashget(vis, f))) {
        ls = ohashget(rev, f);
        ohashput(vis, f, f);
        map append l in ls do {
          <[ops;t]> = ohashget(bbs, l);
          if (not(ops) && length(ohashget(cfg, l))==1) {
            [l]
          } else {add(l); ∅}}};
     if (nxs) loop(nxs, unifiq(get()⊕done))
     else unifiq(get()⊕done)};
  match ret with
    [one] ↦ one
  | else ↦ ∅}
```

There is a common case where a basic block does not contain φ nodes, yet it is a destination of more than one other basic blocks, including empty ones. Those empty blocks can be eliminated.

```
function genssa2_merge_basic_blocks_backwards(src, chgp) {
  //TODO.
  // Remove the basic block if it is empty and if bypassing all of its incoming
  // edges to its destination bb will not cause any phi conflicts (i.e.,
  // either no phis at all or none of the incoming edges are already there).
  src
}
```

```
function genssa2_merge_basic_blocks(src, chgp) {
  // 1. Prepare.
  cfg = mkhash(); cnt = mkhash(); bbs = mkhash(); rev = mkhash();
```

```
getcnt(t) = {
   chk = ohashget(cnt, t);
   if(chk) chk else 0};
addcnt(t) = ohashput(cnt, t, getcnt(t)+1);
visit:genssa2(top: src) {
   once bblock {
      b ↦ {
         ohashput(bbs, name, [ops; t]);
         nxs = genssa2_get_exits(node);
         ohashput(cfg, name, nxs);
         iter n in nxs do {
            ohashput(rev, n, unifiq(name:ohashget(rev, n)))
         };
         iter n in nxs do addcnt(n)}}};

phis = mkhash();
visit:genssa2(top: src) {
  deep bblock {
     b ↦ iter o in ops do o(name)};
  deep oppair: λ(bb) op(bb);
  deep iop(bb) {
     phi ↦ ohashput(phis, bb, bb)
   | else ↦ ∅}};

getnode(nm) = ohashget(bbs, nm);
// 2. Merge basic blocks
collector(add, get) {
  visit:genssa2(top: src) {
    once bblock {
      b ↦ {
         if (not(ohashget(phis, name)) &&
             length(ohashget(rev, name))>1) {
            chk1 = genssa2_paths_converge(cfg, rev, bbs, name);
            if (chk1 && length(ohashget(cfg, chk1))<3)
               add('converge'(chk1, name))};
         nxs = ohashget(cfg, name);
         if (length(nxs)==1 &&
             getcnt(car(nxs)) == 1) {
           add('fuse'(name, car(nxs)))
           }}}};
  commands0 = get();
  // We can only execute independend commands
  ch = mkhash();
  commands = filter [cmd;f;t] in commands0 as {
     if (ohashget(ch,f) || ohashget(ch,t)) ∅
     else {ohashput(ch, f, f); ohashput(ch, t, t); true}};

  if (commands) {
    kill = mkhash(); merge = mkhash();
    iter [cmd; f; t] in commands do {
       ohashput(merge, f, t);
       ohashput(kill, t, f)};
    chgp := true;
    split_phis(bb) =
      collector(add, get)
      collector(phiadd, phiget) {
         visit:genssa2(bblock:bb) {
            deep oppair: op(name);
            deep iop(dstreg) {
               phi ↦ phiadd([dstreg; node])
             | else ↦ add([dstreg; node])}};
         return [phiget();get()]};
```

```
      rewritebb(bb) =
        visit:genssa2(bblock: bb) {
          deep bblock {
            b ↦ { // After merging, phis may need to be
                  //   floated to the top
              <[phiops;nops]> = split_phis(node);
               mk:node(ops = phiops ⊕ nops)}};
          deep phiarg {
            a ↦ {
               chk = ohashget(kill, src);
               if (chk) mk:node(src = chk) else node}}};
      return
      genssa2_kill_dangling(
        visit:genssa2(top: src) {
         deep code: map append b in bs do b;
         deep bblock {
           b ↦ {
              if (ohashget(kill, name)) ∅
              else {
                 chk = ohashget(merge, name);
                 if (chk) {
                   <[ops1;nx1]> = getnode(chk);
                   [rewritebb(mk:node(ops = ops ⊕ ops1, t = nx1))]
                 } else [rewritebb(node)]}}}}})
    } else src}}
```

8 Tree–form representation of expressions

```
ast genssa2tree : genssa2 () {
   expr += op(iop:e)
        |  indvar(ident:id, loopident:l)
        |  rec();}
```

```
function genssa2_describe_simple(env, defs, e) {
   vis = mkhash();
   subst(r0) =
      do loop(r = r0) {
           if (ohashget(vis, r)) 'var'(r)
           else {
             ohashput(vis, r, r);
             df = ohashget(defs, r);
             if (df) {
                   nop = visit:genssa2(iop: df) {
                      deep iop {
                         phi ↦ ∅
                      | call ↦
                            if (genssa2_is_value_pure(env, node)) node
                            else ∅
                      | else ↦ node};
                      deep expr {
                         var ↦ loop(id)
                      | else ↦ node}};
                   if (nop) return 'op'(nop)
                        else return 'var'(r)}
                else 'var'(r)}};
   visit:genssa2(expr: e) {
      deep expr {
           var ↦ subst(id)
        | else ↦ node}}}
```

9 Abstracted algebraic representation of expressions

Some of the passes may benefit from an abstract algebraic form of expressions, provided by the external language interface.
The algebraic language is following:

```
ast genssa2alg {
  aexpr =
      // Arithmetic or alike
        add(srcop:op, aexpr:l, aexpr:r)
      | mul(srcop:op, aexpr:l, aexpr:r)
      | div(srcop:op, aexpr:l, aexpr:r)
      | mod(srcop:op, aexpr:l, aexpr:r)
      | neg(srcop:op, aexpr:l)
      | sub(srcop:op, aexpr:l, aexpr:r)

      // Ordering and comparison
      | eq(srcop:op, aexpr:l, aexpr:r)
      | neq(srcop:op, aexpr:l, aexpr:r)
      | gr(srcop:op, aexpr:l, aexpr:r)
      | ge(srcop:op, aexpr:l, aexpr:r)
      | lt(srcop:op, aexpr:l, aexpr:r)
      | le(srcop:op, aexpr:l, aexpr:r)

      // Abstract
      | additive(srcop:op, aexpr:l, aexpr:r)
      | multiplicative(srcop:op, aexpr:l, aexpr:r)
      | zero(any:c)
      | one(any:c)
      | true(any:c)
      | false(any:c)

      // Flow
      | select(aexpr:c, aexpr:l, aexpr:r)

      // Bail-out
      | fail(expr:e)

      // Atoms
      | var(ident:id)
      | indvar(ident:id)
      | rec()
      | const(any:c)
      ;
}
```

```
function genssa2_to_algebraic(env, v) {
  visit:genssa2tree(expr: v) {
    deep iop {
        select ↦ 'select'(cnd, t, f)
      | call ↦ {
          cls = genssa2_classify(env, dst);
          if (cls) {
            return [cls; dst; @args]
          } else 'fail'('op'(node))}
      | else ↦ 'fail'('op'(node))};
    deep expr {
        op ↦ e
      | var ↦ 'var'(id)
      | indvar ↦ 'indvar'(id)
      | rec ↦ 'rec'()
      | const ↦ 'const'(node)
```

```
    | else ↦ 'fail'(node)}}}
```

```
function genssa2_compop_negate(e) {
  visit:genssa2alg (aexpr: e) {
    once aexpr {
        gr ↦ // !(a>b) = a<=b
           'le'(l,r)
       | ge ↦ // !(a>=b) = a<b
           'lt'(l,r)
       // ... TODO!

       | else ↦ node
    }
  }
}
```

```
function genssa2_from_algebraic(env, av) {
  visit:genssa2alg(aexpr: av) {
    deep aexpr {
      var ↦ 'var'(id)
    | const ↦ c
    | else ↦ ∅ //TODO!
    }}}
```

10 Constant–folding the branch constraints

If a conditional branch is taken, and a condition was a simple function of one register (with all other values being constant, for now, later we can look at following other invariants), and the function is reversible, the value of that register in all the basic blocks dominated by the branch target block can be folded to a constant (or, later, a function of invariants).
We have to solve an equation $A = C$, where A is an arbitrary algebraic expression containing no failures and only one register, and C is a constant. We can get there by iteratively rewriting an equation.

```
function genssa2_is_reversible(env, alg) {
  consteq(v, c, rev) =
  do consteq(v=v,c=c,rev=rev) {
    match v with
      'var'(id) ↦ [id;c]
    | 'add'(op, l, 'const'(c1)) ↦
        consteq(l, 'sub'(∅, c, 'const'(c1)), rev)
    | 'add'(op, 'const'(c1), l) ↦
        consteq(l, 'sub'(∅, c, 'const'(c1)), rev)
    | else ↦ ∅};
  matcheq(a, rev) = {
    match a with
      'eq'(op, x, 'const'(c)) ↦ consteq(x, 'const'(c), rev)
    | else ↦ ∅};
  matchneq(a, rev) = {
    match a with
      'neq'(op, x, 'const'(c)) ↦ consteq(x, 'const'(c), rev)
    | else ↦ ∅};
  do rev(a = alg) {
    match a with
      'eq'(op, x, 'true'(@_)) ↦ matcheq(x, rev)
    | 'eq'(op, x, 'false'(@_)) ↦ matchneq(x, rev)
    | else ↦ ∅
  }}
```

```
function genssa2_algebraic_eligible(env, alg)
collector(vadd, varget) {
   isok = mkref(true);
   visit:genssa2alg(aexpr: alg) {
      deep aexpr {
         var ↦ vadd(id)
       | fail ↦ {isok := ∅}
       | else ↦ ∅}};
   return ^isok && (length(varget())==1)}
```

```
function genssa2_analyse_conditions(env, src, chgp)
collector(cndadd, cndget) {
   // 0. Preparations
  <[cfg;bbs]> = genssa2_cache_cfg(src);
   defs = genssa2_cache_defs(src);
   domtree = graph_dominators(cfg, 'entry');
   domrev = mkhash();
   hashiter(λ(k, vs) {
        iter v in vs do
           ohashput(domrev, v, unifiq(k:ohashget(domrev, v)))
     }, domtree);

   // 1. Collect potentially eligible conditional expressions
   is_simple(e) =
      visit:genssa2tree(expr: e) {
         once expr {
            op ↦ ∅
          | else true}};
   check_condition(bb, c, dsts) = {
      // TODO:
      e0 = genssa2_describe_simple(env, defs, c);
      ae = genssa2_to_algebraic(env, e0);
      if (genssa2_algebraic_eligible(env, ae))
      iter [d;lb] in dsts do {
         chk = genssa2_is_reversible(env, 'eq'(∅, ae, d));
         if (chk) {
           <[reg; avl]> = chk;
            vl = genssa2_from_algebraic(env, avl);
            if (vl) {
              if(is_simple(vl)) {
                cndadd('rewrite'(reg, ∅, lb, vl));
              } else {
                newnm = gensym(); // TODO: meaningful?
                cndadd('rewrite'(reg, newnm, lb, vl))
              }}}}};
   mkconst(e) = visit:genssa2 (expr:e) {
      once expr {
         const ↦ 'const'(node)
       | else ↦
          ccerror('SWITCH-CONDITION-NOT-CONSTANT'(node))}};
   visit:genssa2 (top: src) {
      deep term(bb) {
         brc ↦ check_condition(bb, c, [['true'(∅); tr]; ['false'(∅); fl]])
       | switch ↦ check_condition(bb, v, ns)
       | else ↦ ∅};
      deep switchdst: [mkconst(v); l];
      deep bblock { b ↦ t(name) }};

   // 2. Rewrite condition register values in the dominated blocks
   ndefs = mkhash(); rewrt = mkhash();
   addrw(bb, nm, vl) = {
      ht = {aif (chk  = ohashget(rewrt, bb)) chk
```

```
          else {
            ht = mkhash();
            ohashput(rewrt, bb, ht);
            return ht}};
      ohashput(ht, nm, vl)};
   addrewrite(bb, nm, vl) = {
      dominated = ohashget(domrev, bb);
      iter d in dominated do
         addrw(d, nm, vl)};
   iter rewrite(reg, newnm, bb, vl) in cndget() do
      if (newnm) {
         ohashput(ndefs, bb, [newnm; vl]:ohashget(ndefs, bb));
         addrewrite(bb, reg, 'var'(newnm));
      } else addrewrite(bb, reg, vl);
   rewritebb(ht, bb) =
      visit:genssa2(bblock: bb) {
         deep expr {
            var ↦ {chk = ohashget(ht, id);
                   if(chk) { chgp := true; chk} else node}
         | else ↦ node}};
   return visit:genssa2 (top: src) {
      deep bblock {
        b ↦ {
           nw = ohashget(ndefs, name);
           rwht = ohashget(rewrt, name);
           if (nw) chgp := true;
           if (rwht) rewritebb(rwht, mk:node(ops = nw⊕ops))
           else mk:node(ops = nw⊕ops)}}}}
```

11 Loop invariant motion

Here we're always moving loop invariants outside, without any cost considerations. If a register pressure is becoming an issue, another pass may always do a rematerialisation.

11.1 Utility functions

```
%"A more usable representation of the loop analysis results"
function genssa2_cache_loops(ls) {
  loops = mkhash();
  add(nm, v) = ohashput(loops, nm, v:ohashget(loops, nm));
  iter l in cadr(ls) do {
    match l with
      inaloop(bb, L) ↦
        {add(%Sm<<("body-",L), bb);
         add(%Sm<<("rev-", bb), L)}
    | entryedge(L,f,t) ↦
        {add(%Sm<<("entryedge-",L),[f;t]);
         add(%Sm<<("entry-",L), t)}
    | exitedge(L,f,t) ↦
        {add(%Sm<<("exitedge-",L),[f;t]);
         add(%Sm<<("exit-",L), f)}
    | backedge(L,f,t) ↦
        {add(%Sm<<("backedge-",L),[f;t])}
    | subloop(L1, L2) ↦
        ohashput(loops, %Sm<<("innerloop-", L1, "--", L2), true)
    | else ↦ ∅};
  return loops}
```

```
function genssa2_get_loop_body(loops, L) {
   ohashget(loops, %Sm<<("body-", L))}
```

```
function genssa2_get_loop_entry(loops, L) {
   es = ohashget(loops, %Sm<<("entry-", L));
   match es with
      [one] ↦ one
    | else ↦ ∅}
```

```
function genssa2_get_loop_backedge(loops, L) {
   es = ohashget(loops, %Sm<<("backedge-", L));
   match es with
      [one] ↦ one
    | else ↦ ∅}
```

```
function genssa2_get_loop_exitedge(loops, L) {
   es = ohashget(loops, %Sm<<("exitedge-", L));
   match es with
      [one] ↦ one
    | else ↦ ∅}
```

```
function genssa2_get_loop_entry_edge(loops, L) {
   es = ohashget(loops, %Sm<<("entryedge-", L));
   match es with
      [one] ↦ one
    | else ↦ ∅}
```

```
function genssa2_is_innerloop(loopsht, l, r) {
   ohashget(loopsht, %Sm<<("innerloop-",l,"--",r))}
```

```
function genssa2_is_in_a_loop(loopsht, bb) {
   loops = ohashget(loopsht, %Sm<<("rev-", bb));
   if(loops) car(qsort(λ(l, r) genssa2_is_innerloop(loopsht, l, r), loops))}
```

11.2 Invariant analysis

A register is a loop invariant iff its dependency sub–graph sitting inside a loop is *pure* and only depend on the external variables from the dominators of the loop entry block (excluding the loop entry basic block itself).

```
%"Assuming id is in the innermost loop L,
  check if it is an invariant"
function genssa2_is_a_loop_invariant(
           domtree, deps, loops, L, oright, id, op)
{  iddeps = ohashget(deps, id);
   entry = genssa2_get_loop_entry(loops, L);
   if (entry) { // if more than one entries, retreat
      dset = filter d in ohashget(domtree, entry) as not(d===entry);
      ret = foldl(λ(l,r) l&&r, true,
                  map i in iddeps do
                     aif(chk = ohashget(oright, i))
                         memq(chk, dset)
                     else true);
      if(ret) return genssa2_get_loop_entry_edge(loops, L) else ∅}}
```

```
function genssa2_find_loop_invariants(env, src, loops) {
   // 1. Preparations
  <[cfg;bbs]> = genssa2_cache_cfg(src);
   defs = genssa2_cache_defs(src);
   domtree = graph_dominators(cfg, 'entry');
   oright = genssa2_cache_origs(src);
   deps = genssa2_make_depgraph(src, ∅);
   loopsht = genssa2_cache_loops(loops);

   // 2. Find candidates
   collector(addinvariant, getinvariants) {
     visit:genssa2(top: src) {
      deep bblock {
        b ↦ {
          // Find the innermost loop for this bb
          loopnest = genssa2_is_in_a_loop(loopsht, name);
          // Do not even bother if it's not in a loop
          if (loopnest) iter o in ops do o(name, loopnest)}};
       deep oppair: λ(bb, L) {
          if (op) {
             chk = genssa2_is_a_loop_invariant(domtree, deps, loopsht,
                     L, oright, name, op);
             if (chk) addinvariant([bb;name;op;L;chk])}};
       deep iop {
          phi ↦ ∅ // cannot be an invariant
        | select ↦ node // maybe
        | call ↦ if (genssa2_is_pure(env, dst)) node else ∅}};
     inv = getinvariants();
     if(inv) {
       return inv
     }}}
```

```
function genssa2_move_loop_invariants(src, invariants) {
   // Prepare invariants
   invariantsht = mkhash(); edges = mkhash();
   iter [bb;name;op;L;ft] in invariants do {
      ohashput(edges, bb, ft);
      ohashput(invariantsht, bb, ohashget(invariantsht, bb)⊕[[name;op]])};

   // Loop invariants are relocated into new intermediate basic blocks that
   // are injected into the entry edges
   newedges = mkhash();
   killed = mkhash();

   injected =
     hashmap(λ(name, ops) {
       <[f;t]> = ohashget(edges, name);
        iter [dst;op] in ops do {
           ohashput(killed, dst, dst)};
        edg = gensym();
        ohashput(newedges, %Sm<<(f,'--',t), edg);
        return 'b'(edg, ops, 'br'(t))}, invariantsht);


   rewrite_edge(f, t) = ohashget(newedges, %Sm<<(f, '--', t));
   delete_op(id) = ohashget(killed, id);
   rewrite_phis(bb, ops) =
      map o in ops do
         visit:genssa2(oppair: o) {
            deep phiarg {
               a ↦
                 aif (chk = rewrite_edge(src, bb))
```

```
                  mk:node(src = chk)
                else node }};
  rewrite_term(t, bb) =
    visit:genssa2(term: t) {
      deep labident:
        aif(chk = rewrite_edge(bb, node)) chk
           else node};
  visit:genssa2(top: src) {
    deep code: bs⊕injected;
    deep bblock {
      b ↦ { ops1 = map append ops do ops;
            mk:node(ops = rewrite_phis(name, ops1),
                    t = rewrite_term(t, name))}};
    deep oppair: if (delete_op(name)) ∅ else [node]}}
```

```
function genssa2_loop_invariants(env, src, chgp) {
  loops = genssa2_loops(src);
  inv = genssa2_find_loop_invariants(env, src, loops);
  if(inv) {
    chgp := true;
    return genssa2_move_loop_invariants(src, inv)
  } else src}
```

12 Induction variables

A variable is an induction variable of a loop L iff:

- It is a φ node located in the loop entry node, with only two entries
- There is a circular dependency on itself
- All of the circular dependency path is within the loop L (it does not matter if there is another circular dependency in an outer loop)
- All of the dependency path components are pure (otherwise no further analysis is useful) and no φs, with only the entry edge one being a φ. The only exception for this rule is an inferior induction variable with computable bounds, but this case is a big fat TODO.

Once the determine that this is an induction variable we're recording it as follows:
`inductive(loop-id, reg, path, [loop-nodes], init, step)`.
We should mark induction variables starting from the innermost loops.

```
function genssa2_describe_induction_step(env, defs, reg,
               depg, entry, revh)
{ vis = mkhash();
  subst(r0) =
    do loop(r = r0) {
     if ( r === reg ) return 'rec'()
     else {
       df = ohashget(defs, r);
       if (not(df)) return 'var'(r)
       else if (ohashget(vis, r)) return 'var'(r)
       else {
         ohashput(vis, r, r);
         nop = visit:genssa2(iop: df) {
                deep expr {
                  var ↦ {aif (chk0 = ohashget(revh, id))
                             'indvar'(id, chk0)
                          else if (ohashget(depg, id)) loop(id)
                          else node}
                | else ↦ node}};
         return 'op'(nop)}}};
  visit:genssa2(iop: ohashget(defs, reg)) {
```

```
    deep phiarg {
       a ↦ if (src === entry) ∅ else v};
    deep iop {
       phi ↦ {match (filter a in args as a) with
                   [one] ↦ one
                 | else ↦ ccerror('OOPS'(args))}
     | else ↦ ccerror('IMPOSSIBLE'(node))};
    deep expr { var ↦ if (ohashget(depg, id)) subst(id) else node
              | else ↦ node}}}
```

```
function genssa2_maybe_induction(env, loops, defs, deps, origs, src)
collector(add, get) {
 //  In order to do so we'd have to postpone purity check and
 //  do it in a loop, eliminating inner inductive variables one by one.

   innerloop(bb) = {
      // Return a name of the innermost loop for this node, or []
      return genssa2_is_in_a_loop(loops, bb)};

   getloopnodes(L) = {
      // Cache all the nodes of a loop L
      l = ohashget(loops, %Sm<<("body-", L));
      ht = mkhash();
      iter l do ohashput(ht, l, l);
      return ht};

   isinaloop(lh, reg) =
      aif (chk = ohashget(origs, reg))
          ohashget(lh, chk);

   follow_deps(lh, entry) = {
      // Only follow the register dependencies that lie in
      // a given loop (lh is a hashtable).

      ht = mkhash();
      do loop(e = entry) {
         if (ohashget(ht, e)) ∅
         else {
            refs = ohashget(deps, e);
            lrefs = filter r in refs as isinaloop(lh, r);
            ohashput(ht, e, lrefs);
            iter r in lrefs do loop(r)}};
      return ht};

   iscircular(ht, entry) = {
      // Check if there are circular dependencies in a graph
      vis = mkhash();
      do loop(e = entry) {
         if (ohashget(vis, e)) true
         else {
           ohashput(vis, e, e);
           do iloop(r = ohashget(ht, e)) {
              match r with
                hd:tl ↦ if(loop(hd)) true else iloop(tl)
              | else ↦ ∅}}}};

   ispure(reg, r) = {
      // Check if the register definition is pure and not a phi
      if (r === reg) true
      else {
        chk = ohashget(defs, r);
        visit:genssa2(iop: chk) {
```

```
      deep iop {
         phi ↦ ∅
        | select ↦ true
        | call ↦ genssa2_is_value_pure(env, chk)
        | else ↦ ∅}}}};

isnpure(reg, k) = genssa2_is_value_pure(env, ohashget(defs, k));
isnnpure(rev, reg, k) = {
   if (ohashget(rev, k)) true
   else ispure(reg, k)};

loopdescr(L) = {
   // TODO: may want more information here
   ohashget(loops,%Sm<<("body-",L));
};

getloopentryedge(L) = {
   chk = ohashget(loops, %Sm<<("entryedge-", L));
   match chk with
     [[f;t]] ↦ f
     //TODO!
   | else ↦ ccerror('NOENTRYEDGE'(L))};

initvalue(L, reg) = {
   edge = getloopentryedge(L);
   visit:genssa2(iop: ohashget(defs, reg)) {
      deep phiarg {
         a ↦ if (src===edge) v else ∅};
      deep iop {
         phi ↦ {r = filter a in args as a;
                match r with [one] ↦ one
                           | else ↦ ccerror('NO-ENTRY'(reg))}
      | else ↦ ccerror('IMPOSSIBLE'(node))}}};

makestep(L, reg, depg, revh) = {
   // TODO
   entry = getloopentryedge(L);
   genssa2_describe_induction_step(env, defs, reg, depg, entry, revh)};

tryinduct(L, reg) = {
   // 1. Build a sub-graph of the reg dependency graph
   //    which lies entirely within L.
   Lnodes = getloopnodes(L);
   depg = follow_deps(Lnodes, reg);
   // 2. If it is still circular, check if all its elements
   //    are pure.
   if (iscircular(depg, reg)) {
      depl = hashmap(λ(k,v) k, depg);
      purep = foldl(λ(l,r) l&&r, true,
                    map d in depl do ispure(reg, d));
   //  3. Still here? Add an 'inductive' node.
      if (purep) {
         add('inductive'(L, reg, loopdescr(L), depl,
                     'init'(genssa2_to_algebraic(env, initvalue(L, reg))),
                     genssa2_to_algebraic(env, makestep(L, reg, depg, mkhash()))))
      } else {
        npurep = foldl(λ(l,r) l&&r, true,
                    map d in depl do isnpure(reg, d));
        if(npurep) add('maybeinductive'(L, reg, depl))}}};

refineinduct(lst) =
   do loop(l = lst) collector(iadd, iget) {
      chgp = mkref(∅); rev = mkhash();
```

```
        todo = map append l in lst do {
          match l with
            'inductive'(LN, r, LPth, pth, @_) ↦ {
              iadd(l); ohashput(rev, r, LN); ∅}
          | 'maybeinductive'(L, reg, nppth) ↦ [l]};
        nxt = map append t in todo do {
          match t with
             'maybeinductive'(L, reg, nppth) ↦ {
               chk = foldl(λ(l, r) l&&r, true,
                           map p in nppth do isnnpure(rev, reg, p));
               if(chk) {
                  chgp := true;
                  depg = mkhash();
                  iter d in nppth do ohashput(depg, d, d);
                  iadd('inductive'(L, reg, loopdescr(L), nppth,
                              'init'(genssa2_to_algebraic(env, initvalue(L, reg))),
                              genssa2_to_algebraic(env, makestep(L, reg, depg, rev))));
                  return ∅}
               else return [t]}};
        if (nxt && ^chgp) loop(iget()⊕nxt) else iget()};

    visit:genssa2(top: src) {
      deep iop(bb, dstreg) {
        phi ↦ {
           L = innerloop(bb);
           if (L && length(args)==2) { // In da loop
              tryinduct(L, dstreg)}}
      | else ↦ ∅};
      deep oppair: λ(bb) op(bb, name);
      deep bblock {b ↦ iter o in ops do o(name)}};

    candidates = get();
    // Now, iterate over candidates until there are no more changes
    return refineinduct(candidates)}
```

```
function genssa2_induction(env, src) {
   loops = genssa2_loops(src);
   defs = genssa2_cache_defs(src);
   origs = genssa2_cache_origs(src);

   deps = genssa2_make_depgraph(src, ∅);
   loopsht = genssa2_cache_loops(loops);

   l = genssa2_maybe_induction(env, loopsht, defs, deps, origs, src);
   return [[loops; loopsht; defs; origs; deps]; l]
}
```

13 Loop exit conditions analysis

If a loop exit condition is a function of loop induction variables and loop invariants (or constants) only, we can check it for certain patterns:

- $cmp(L_i, I)$ — gives a bound for a linear induction variable, which may later be used to fold a L_{ind} outside of the loop and be used to define value set inside the loop.
- $eq(X_i, I)$ — folds to I outside of a loop, no matter how X_i steps.

```
function genssa2_loop_exits(env, cache, ind, src)
collector(addexit, getexits) {
   // 1. Get loop exit conditions
  <[[lx;loops]; loopsht; defs; origs; deps]> = cache;
```

```
    exits = mkhash();
    iter l in loops do match l with
      exitedge(L,f,t) ↦ { ohashput(exits, f, [L;t]); };

  writeline('IND'(@ind));
    indht = mkhash();
    iter i in ind do
       match i with
         inductive(L, reg, ld, depl, init, step) ↦ {
           ohashput(indht, reg, L)};

    describe(L, c) = {
      lbody = ohashget(loopsht, %Sm<<("body-", L));
      vis = mkhash();
      subst(r0) =
        do loop(r = r0) {
          chk1 = ohashget(indht, r);
          if (ohashget(vis, r)) 'var'(r)
          else if (chk1) {
            return 'indvar'(r, chk1)
          } else {
            ohashput(vis, r, r);
            df = ohashget(defs, r);
            if (df) {
              o = ohashget(origs, r);
              if (memq(o, lbody)) {
                nop = visit:genssa2(iop: df) {
                  deep iop {
                    phi ↦ ∅
                   | else ↦ node};
                  deep expr {
                    var ↦ loop(id)
                   | else ↦ node}};
                if (nop) return 'op'(nop)
                   else return 'var'(r)}
              else 'var'(r)}
            else 'var'(r)}};
     visit:genssa2(expr:c) {
        deep expr {
          var ↦ subst(id)
         | else ↦ node}}};

    describe_alg(L,c) = genssa2_to_algebraic(env, describe(L, c));
    visit:genssa2(top: src) {
      deep bblock { b ↦ t(name) };
      deep term(bb) {
        brc ↦ {
          chk = ohashget(exits, bb);
          match chk with
            [L;ex] ↦
              {neg = (fl === ex);
               addexit('exit'(bb, ex, L, neg, describe_alg(L, c)))}}
      | else ↦ ∅}};
    exits = getexits();
    return exits
  }
```

Useful explanations for induction variables and exit conditions include the following patterns:

- $i_0 = C_0, i \leftarrow i + C_1, [i < C_2]$ — a constant bound loop, with value of i known on a loop exit, and a total number of iterations being a known constant (affecting the exit values of all the other induction variables). There must be only one exit in such a loop.

```
function genssa2_loop_bounds(env, ind, exits)
collector(addcandidate, getcandidates) {
   // Select the loops with one exit only, which is matching a pattern
   //       (compop (indvar X) (const ...))
   //    or
   //       (compop (const ...) (indvar X))
   //
   //    And indvar X definition is matching a pattern
   //       ((init (const ...)) (step (arithmetic (rec) (const ...))))
   //    or
   //       ((init (const ...)) (step (arithmetic (const ...) (rec))))

   // 1. Cache the candidate constant induction variables
   indht = mkhash();
   addind(id, ic, stepop, dir, sc) = ohashput(indht, id, [ic; stepop; dir; sc]);
   iter i in ind do {
     match i with
       'inductive'(L, regnm, loopnodes, deps, 'init'('const'(ic)),
                    [stepop; op1; opL; opR]) ↦ {
          match [opL; opR] with
             ['rec'();'const'(sc)] ↦ addind(regnm, 'const'(ic), stepop, 'lr', 'const'(sc))
           | ['const'(sc);'rec'()] ↦ addind(regnm, 'const'(ic), stepop, 'rl', 'const'(sc))}};

   // 2. Count the exits
   exitsht = mkhash();
   iter e in exits do {
     match e with
       'exit'(bb,ex, L, @_) ↦ ohashput(exitsht, L, bb:ohashget(exitsht, L))};

   // 3. Check the candidate constant exit conditions
   isconstind(V) = ohashget(indht, V);
   iscmpop(op) =
     case op { 'eq' | 'neq'| 'gr' | 'ge' | 'lt' | 'le' ↦ true | else ↦ ∅};

   makeop(neg, dir, op, c) = {
     <L:R> = if(dir==='lr') 'rec'():c else c:'rec'();
      ret = [op; '_'; L; R];
      if (not(neg)) return genssa2_compop_negate(ret) else ret};

   makestepop(op, dir, c) = {
     <L:R> = if(dir==='lr') 'rec'():c else c:'rec'();
     return [op; '_'; L; R]};

   makebounds(L, neg, dir, op, V, c) = {
     <[ic;stepop;sdir;sc]> = ohashget(indht, V);
      addcandidate('constloop'(L, V, 'init'(ic), 'step'(makestepop(stepop, sdir, sc)), 'bound'(makeop(neg, dir, op, c
   };

   check(L, neg, dir, op, V, c) =
    if(length(ohashget(exitsht,L))==1) {
       if(iscmpop(op) && isconstind(V)) {
         makebounds(L, neg, dir, op, V, c)
       }};
   iter e in exits do {
     match e with
       'exit'(bb,ex, L, neg, [op;op1;'indvar'(V);'const'(c)]) ↦
           check(L, neg, 'lr', op, V, 'const'(c))
    |  'exit'(bb,ex, L, neg, [op;op1;'const'(c);'indvar'(V)]) ↦
           check(L, neg, 'rl', op, V, 'const'(c))};
   return getcandidates();
}
```

```
function genssa2_loop_bounds_static(env, lb) {
  // Further simplify the static loop bounds, infer the precise integer intervals possible
  getinteger(c) = match c with const(c0) ↦
                    genssa2_env_getinteger(env, c0)
                  | else ↦ ccerror('IMPOSSIBLE'(c));
  normalise_step(op,L,R) = {
    match [op;L;R] with
       add(rec(),x) ↦ getinteger(x)
     | add(x,rec()) ↦ getinteger(x)
     | sub(rec(),x) ↦ 0-getinteger(x)
     | else ↦ ∅};
  max(a,b) = if(a>b) a else b;
  min(a,b) = if(a<b) a else b;
  abs(a) = if(a>0) a else 0-a;
  normalise_limit(initc, step, op, L, R) =
  do loop(op=op, L=L, R=R) {
    match [op;L;R] with
       le(rec(),x) ↦ {
         // TODO: check off intervals
         c1 = getinteger(x);
         d = c1 - initc;
         n = d / step;
         if (initc + d*n == c1) return (c1+1) else return c1}
     | lt(rec(),x) ↦ {
         c1 = getinteger(x);
         return c1}
     | gr(rec(),x) ↦ {
         cl = getinteger(x);
         return cl}
     | ge(rec(),x) ↦ {
         c1 = getinteger(x);
         d = initc - c1;
         n = d / abs(step);
         if (initc - d*n == c1) return (c1+1) else return c1}
     | le(x,rec()) ↦ loop('ge','rec'(),x)
     | lt(x,rec()) ↦ loop('gr','rec'(),x)
     | ge(x,rec()) ↦ loop('le','rec'(),x)
     | gr(x,rec()) ↦ loop('lt','rec'(),x)
       // TODO
     | else ↦ ∅};
  normalise_count(i,s,l) = {
     nsteps = (max(i,l) - min(i, l)) / abs(s);
     return nsteps
  };
  map append constloop(Lp,V,init(I),step([op;_;L; R]),bound([bop;_;bL;bR])) in lb do {
    initc = getinteger(I);
    istep = normalise_step(op,L,R);
    if(istep) {
      blimit = normalise_limit(initc, istep, bop, bL, bR);
      if (blimit) {
         count = normalise_count(initc, istep, blimit);
        ['constloop'(Lp,V, initc, istep, blimit, count)]
      }}}}
```

14 Loop unrolling

Once again, we're not after any performance optimisations here, so we can do the simplest thing possible and rely on a number of consequent passes to clean up the mess.

The simplest loop unrolling, especially if we know the static bounds, is to copy the loop basic blocks N times (doing all the renaming required, of course), replacing the back edge with the next edge, and induction phi with only its back edge branch.

```
function genssa2_loop_unroll_inner(env, src, bbsh, loop, loopbackedge, loopexitedge, count, chgp) {

   newnames(loopbbs, entrybb, newentrybb) = {
     bht = mkhash();rht = mkhash();
     addbbrename(a,b) = ohashput(bht, a, b);
     addregrename(a,b) = ohashput(rht, a, b);
     visit:genssa2 (code: loopbbs) {
        deep bblock {
           b ↦
              if (name === entrybb)
                 addbbrename(name, newentrybb)
              else addbbrename(name, gensym())};
        deep oppair: addregrename(name, gensym())};
     return bht:rht};

   rewrite(loopbbs, names, prevnames, backedge, exitedge, firstp, lastp, nextname) = {
    <bht:rht> = names;
    <pbht:prht> = prevnames;
     isloopbb(name)  = ohashget(bht, name);
     renamebb(name)  = aif(r = ohashget(bht, name)) r else name;
     renamereg(name) = aif(r = ohashget(rht, name)) r else name;
     prevrenamebb(name) = ohashget(pbht, name);
     prevrenamereg(name) = aif(r = ohashget(prht, name)) r else name;
     isbackedge(f, t) = (%Sm<<(f, "->", t) === backedge);
     isexitedge(f, t) = (%Sm<<(f, "->", t) === exitedge);
     rewriteexpr(e) =
       visit:genssa2 (expr: e) {
       deep expr {
          var ↦ mk:node(id = renamereg(id))
        | else ↦ node}};
     prevrewriteexpr(e) =
       visit:genssa2 (expr: e) {
       deep expr {
          var ↦ mk:node(id = prevrenamereg(id))
        | else ↦ node}};
     rewriteterm(t) =
       visit:genssa2 (term: t) {
          deep labident: renamebb(node);
          once expr { else ↦ rewriteexpr(node) }};
     rewritephi(bbname, regname, args) = {
       newargs = map append a in args do
          visit:genssa2 (phiarg: a) {
             deep phiarg {
               a ↦ if (firstp && isbackedge(src, bbname)) ∅
                    else if (isbackedge(src, bbname)) ['a'(prevrenamebb(src), prevrewriteexpr(v))]
                    else if (not(firstp) && not(isloopbb(src))) ∅
                    else if (firstp && not(isloopbb(src))) ['a'(src, v)]
                    else ['a'(renamebb(src), rewriteexpr(v))]}};
       return 'phi'(@newargs)};
     visit:genssa2 (code: loopbbs) {
        deep bblock {
           b ↦ {newnm = renamebb(name);
                mk:node(name = newnm, ops = map o in ops do o(name), t = t(name))}};
        deep oppair: λ(bbname) {
           [renamereg(name); op(bbname, name)]};
        once iop {
           phi ↦ λ(bbname, regname) rewritephi(bbname, regname, args)
         | deep ↦ { else ↦ λ(bbname, regname) node }};
        once term(bbname) {
           brc ↦ if(isbackedge(bbname, tr)) 'br'(nextname)
                 else if(isbackedge(bbname, fl)) 'br'(nextname)
                 else if(isexitedge(bbname, fl) && lastp) rewriteterm('br'(fl))
                 else if(isexitedge(bbname, tr) && lastp) rewriteterm('br'(tr))
```

```
                else if(isexitedge(bbname, fl)) rewriteterm('br'(tr))
                else if(isexitedge(bbname, tr)) rewriteterm('br'(fl))
                else rewriteterm(node)
         | br  ↦ if(isbackedge(bbname, dst)) 'br'(nextname)
                else rewriteterm(node)
         | else ↦ rewriteterm(node)};
         once expr { else ↦ rewriteexpr(node) }}};

  rewriteloop(loopbbs, entrybb, count, backedge, exitedge, exitname) = collector(addbb, getbbs) {
   ret = do loop(prevnames = (mkhash():mkhash()),
             prevnext = entrybb, nextname0 = gensym(),
             counter = 0) {
      nextname = if(counter==count) exitname else nextname0;
      nnm = newnames(loopbbs, entrybb, prevnext);
      newbbs = rewrite(loopbbs, nnm, prevnames, backedge, exitedge,
                    counter==0, counter==count, nextname);
      iter b in newbbs do addbb(b);
      if (counter < count)
        loop(nnm, nextname, gensym(), counter+1)
      else nextname:nnm
   };
   return ret:getbbs()};

   loopbbs = map l in loop do ohashget(bbsh, l);
   loopht = mkhash(); iter l in loop do ohashput(loopht, l, l);
  <[bf;entrybb]> = loopbackedge;
   backedge = %Sm<<(bf, "->", entrybb);
  <[ef;et]> = loopexitedge;
   exitedge = %Sm<<(ef, "->", et);
  <(nextname:(bht:rht)):newbbs> = rewriteloop(loopbbs, entrybb, count, backedge, exitedge, et);
   renamereg(id) = aif(chk = ohashget(rht, id)) chk else id;
   renamebb(id)  = aif(chk = ohashget(bht, id)) chk else id;
   chgp := true; // TODO - there must be bail out conditions, right?
   (map append bb in src do
     visit:genssa2(bblock: bb) {
       deep bblock {
         b ↦ if(ohashget(loopht, name)) ∅
              else [node]};
       deep expr {
         var ↦ mk:node(id = renamereg(id))
        | else ↦ node};
       deep phiarg {
         a ↦ if (ohashget(loopht, src)) mk:node(src = renamebb(src)) else node}}) ⊕ newbbs}
```

```
function genssa2_loop_unroll(env, src, cache, lr, chgp) {
 <[[_;_]; loops; @_]> = cache;
 <constloop(Lp, V, initc, istep, blimit, count)> = lr;
  if (count > 16) src else
  {
    loopbackedge = genssa2_get_loop_backedge(loops, Lp);
    loopexitedge = genssa2_get_loop_exitedge(loops, Lp);
    loop = genssa2_get_loop_body(loops, Lp);
    bbsh = mkhash();
    srcc = visit:genssa2 (top: src) {
      once top { f ↦ body }};
    visit:genssa2(code: srcc) {
      once bblock { b ↦ ohashput(bbsh, name, node) }};

    nbody = genssa2_loop_unroll_inner(env, srcc, bbsh, loop, loopbackedge, loopexitedge, count, chgp);
    genssa2_kill_dangling(visit:genssa2 (top:src) {
      once top { f ↦ mk:node(body = nbody) }})}}
```

15 Generic CSE pass

```
function genssa2_normalise_const(env, t, v) {
  //TODO!
  return %S<<(v)}
```

```
function genssa2_normalise_other(env, t, v) {
  //TODO!
  return %S<<(v)}
```

```
function genssa2_cse(env, src, modp) {
  // *. Utility
  intersect(a, b) = collector(add, get) {
     ht = mkhash();
     iter a do ohashput(ht, a, a);
     iter b do if(ohashget(ht, b)) add(b);
     return get()};
  mapintersect(l) = foldl(intersect, car(l), cdr(l));
  // TODO: share with some other pass?
 <[cfg;bbs]> = genssa2_cache_cfg(src);
  domtree = graph_dominators(cfg, 'entry');
  defs = genssa2_cache_defs(src);
  defbbs = genssa2_cache_origs(src);
  depgraph = genssa2_make_depgraph(src, true);
  revdeps = mkhash();
  hashiter(λ(k, vs) {
       iter v in vs do ohashput(revdeps, v, unifiq(k:ohashget(revdeps, v)))
    }, depgraph);

  // 1. Collect equivalent assignments (i.e., same function or select applied to
  //     exactly the same arguments, using the model to check for the constant equivalence).
  normalise(e) =
    visit:genssa2(expr:e) {
     once expr {
        var ↦ %S<<("var-", id)
      | glob ↦ %S<<("glob-", id)
      | const ↦ genssa2_normalise_const(env, t, v)
      | other ↦ genssa2_normalise_other(env, t, v)}};
  newgroup() = mkhash();
  groups = mkhash();
  getgroup(id) = {
     aif(chk = ohashget(groups, id)) chk
     else {
        grp = newgroup();
        ohashput(groups, id, grp);
        return grp}};
  getarggroup(group, args) = {
     flat = strinterleave(args, " | ");
     aif(chk = hashget(group, flat)) chk
     else {
        nw = mkref(∅);
        hashput(group, flat, nw);
        return nw}};
  addreg(group, reg) = {group := reg : (^group)};
  collect_call(reg, fn, args) = {
    fngroup = getgroup(fn);
    nargs = map a in args do normalise(a);
    group = getarggroup(fngroup, nargs);
    addreg(group, reg)};
  collect_select(reg, cnd, t, f) = {
```

```
  fngroup = newgroup();
  nargs = map a in [cnd; t; f] do normalise(a);
  group = getarggroup(fngroup, nargs);
  addreg(group, reg)};
visit:genssa2 (top: src) {
  deep oppair: op(name);
  deep iop(reg) {
     phi ↦ ∅
   | select ↦ collect_select(reg, cnd, t, f)
   | call ↦ if (genssa2_is_pure(env, dst)) collect_call(reg, dst, args)}};

lgroups = collector(addg, getgs) {
   hashiter(λ(k, ht) {
        hashiter(λ(kk, v) {
                    x = ^v; if (length(x)>1) addg(reverse(x))},
                 ht)},
        groups);
   getgs();
};

// 2. For each group of equivalent assignments, replace them all with a single assignment located
//    at the nearest common dominator (see the same thing in the loop invariant code motion implementation).
//    Rename all the instances.
//    In order to avoid confusion, eliminate all the groups that are directly dependent on any of the
//    assignments that are being moved. We're not after any performance optimisations here. Those groups
//    will be taken care of at the next iteration anyway.
revtree = mkhash();
hashiter(λ(k,v) {
   iter v do ohashput(revtree, v, k:ohashget(revtree, v))}, domtree);

removeht = mkhash(); renameht = mkhash(); newht = mkhash();

addnew(bb, cg, v) = ohashput(newht, bb, [cg;v]:ohashget(newht, bb));
addremove(g) = ohashput(removeht, g, g);
addrename(f, t) = ohashput(renameht, f, t);

doremove(id) = ohashget(removeht, id);
dorename(id) = aif(chk = ohashget(renameht, id)) chk else id;
getnew(bb) = ohashget(newht, bb);

tainted = mkhash();
iter g in lgroups do {
   // 0. Get the list of dependencies, bail out if they're tainted. Taint the entire group;
   deps0 = ohashget(depgraph, car(g));
   deps = map append d in deps0 do if(ohashget(defbbs, d)) [d] else ∅;
   t = foldl(λ(x,y) (x||y), ∅, map d in deps do ohashget(tainted, d));
   iter g do ohashput(tainted, g, g);
   if (not(t)) {
     // 1. List bblocks for all the uses and dependencies in this group
     uses = unifiq(map append g do ohashget(revdeps, g));
     usebbs = unifiq(map u in uses do ohashget(defbbs, u));
     depbbs = unifiq('entry':map d in deps do ohashget(defbbs, d));
     // 2. Find a basic block dominated by all the dependencies of this assignment,
     //    and dominating all the blocks in the list.
     dominated  = mapintersect(map d in depbbs do ohashget(revtree, d));
     dominators = mapintersect(map d in usebbs do ohashget(domtree, d));
     here = intersect(dominated, dominators);
     if (here) {
       modp := true;
       targetbb = car(here);
       // 3. Move an assignment there, preserving the name of the first assignment in this group;
       //    Add all other assignments to a rename table.
       //    Do not move a declaration if it is already in the target basic block (obviously).
```

```
    <cg:rst> = g;
     if (not(ohashget(defbbs, cg) === targetbb)) {
        addnew(targetbb, cg, ohashget(defs, cg));
        addremove(cg);
     };
     iter r in rst do {
        addrename(r, cg);
        addremove(r)};
   }}};
// 3. Commit the changes
if(lgroups)
  visit:genssa2 (top: src) {
    deep bblock {
      b ↦
         mk:node(ops = (map append o in ops do o)⊕getnew(name))};
    deep oppair : {
      if (doremove(name)) ∅ else [node]};
    deep expr {
      var ↦ mk:node(id = dorename(id))
    | else ↦ node}}
else src}
```

16 Generic type propagation

User must provide typing rules for each intrinsic. E.g.:

$$x = add(l, r) \to \{T_x = T_l = T_r\}.$$

This can be used for propagating types missing from GEP and φ nodes, as well as for typing all the nodes.

```
function genssa2_intrinsic_equations(lenv, env, eqadd, dstreg, dst, args) {
  eqnmaker = ohashget(env, '*type-equation-maker*');
  if (eqnmaker) {
     eqn = eqnmaker(dst);
     if (eqn) eqn(lenv, eqadd, dstreg, dst, args)}}
```

```
function genssa2_make_boolean_type(env) {
  btype = ohashget(env, '*boolean-type*');
  if (btype) btype else 'var'('boolean')}
```

```
function genssa2_make_type_maker(lenv, env) {
  tmaker = ohashget(env, '*type-maker*');
  if (tmaker) λ(t) tmaker(lenv, t) else λ(tp) 'var'(gensym())}
```

```
function genssa2_make_ctype_maker(lenv, env) {
  tmaker = ohashget(env, '*ctype-maker*');
  if (tmaker) λ(t, c) tmaker(lenv, t, c) else λ(tp,vl) 'var'(gensym())}
```

```
function genssa2_type_equations(env, lenv, src)
collector(eqadd, eqsget) {
  booltp = genssa2_make_boolean_type(env);
  mktype = genssa2_make_type_maker(lenv, env);
  mkctype = genssa2_make_ctype_maker(lenv, env);
  aeq(l, r) = eqadd('equals'(l, r));
  aeqv(l, r) = eqadd('equals'('var'(l), r));
```

```
  visit:genssa2(top: src) {
    deep top { f ↦ aeqv('*return*', mktype(ret)) };
    deep argpair: aeqv(name, mktype(t));
    deep oppair: op(name);
    deep phiarg(dstreg) { a ↦ aeqv(dstreg, v)};
    deep switchdst: v;
    deep iop(dstreg) {
       phi ↦ iter a in args do a(dstreg)
     | select ↦ {aeqv(dstreg, t); aeqv(dstreg, f)}
     | call ↦ genssa2_intrinsic_equations(lenv, env, eqadd, dstreg, dst, args)};
    deep term {
       brc ↦ aeq(c, booltp)
     | switch ↦ iter n in ns do aeq(v, n)
     | else ↦ ∅};
    deep expr {
       var ↦ 'var'(id)
     | glob ↦ 'var'(id)
     | const ↦ mktype(t,v)
     | other ↦ mktype(t,v)}};
  return eqsget()}
```

```
function genssa2_eqn_to_prolog(e0)
  do loop(e = e0) {
    match e with
      'var'(nm) ↦ 'var'(nm)
      // A clumsy way to separate 'foo()' from 'foo'
    | [one] ↦ 'term'(%Sm<<("tp__",one), 'term'('tpx__dummy'))
    | 'equals'(l,r) ↦ 'term'('equals', loop(l), loop(r))
    | [tg;@args] ↦ 'term'(%Sm<<("tp__",tg), @map a in args do loop(a))
    | else ↦ 'term'(%Sm<<("tp__", e))}
```

```
parser prologtmp (prologlex) {
   prologtmp ⇐ {"tp__" [constident]:id ⇒ id}
            / {[constident]:id ⇒ wtf(id)};}
```

```
function genssa2_prolog_to_type(d) {
  s(id) =
    { t = %S<<(prolog_strip_id(id));
      parse t as prologtmp};

  do loop(e = d) {
    match e with
       'var'(id) ↦ '*type-var*'(id)
     | 'term'(id, 'term'('tpx__dummy/0')) ↦ [s(id)]
     | 'term'(id) ↦ s(id)
     | 'term'(id, @args) ↦ [s(id);@map(loop, args)]}}
```

```
function genssa2_solve_type_equations(env, lenv, eqns) {
   prolog = map e in eqns do genssa2_eqn_to_prolog(e);
   result = prolog_backend_driver(
              ['query'(@prolog)],
              %read-compile-eval ,
              ∅);
   match result with
     num:varmap:nextfn ↦ {
       dvars = map [nm;d] in varmap do {
          return [nm; genssa2_prolog_to_type(d)]};
       ht = mkhash();
```

```
        iter [nm;d] in dvars do ohashput(ht, nm, d);
        return ht}
   | else ↦ mkhash() // Failed to type
}
```

17 Interface

```
%"All the abstract SSA steps together"
function genssa2_process_iter(modenv, src0) {
   do loop(src = src0) {
      chgp = mkref(∅);

      // 1. Iterative fold and kill.
      src1 = genssa2_fold_and_kill(modenv, src, chgp);
      // 2. Handle the loose phi loops
      src2 = genssa2_dce_phi(modenv, src1, chgp);
      // 3. Rewrite eligible phis as selects
      src3 = genssa2_detect_selects(modenv, src2, chgp);
      // 4. Clean up CFG
      src4x = genssa2_merge_basic_blocks(src3, chgp);
      src4  = genssa2_merge_basic_blocks_backwards(src4x, chgp);

      // 5. Move loop invariants
      src5 = genssa2_loop_invariants(modenv, src4, chgp);

      println("STEP5");
      printgenssa2(src5);
      println("--------------");
     <[cache;inds]> = genssa2_induction(modenv, src5);
      src5x = if (inds) {
         exits = genssa2_loop_exits(modenv, cache, inds, src5);
         cnds = genssa2_loop_bounds(modenv, inds, exits);
         cnds = genssa2_loop_bounds_static(modenv, cnds);
         if(cnds) {
                  // TODO: only unroll if hinted
                  return genssa2_loop_unroll(modenv, src5, cache,
                                             car(cnds), chgp)
               }
         else return src5
      } else src5;
      src6 = genssa2_analyse_conditions(modenv, src5x, chgp);

      // 6. CSE
      src7 = genssa2_cse(modenv, src6, chgp);

      // TODO:
      //
      // - Branch constraint folding
      // - GVN (do we need it? CSE is there already)
      // - Loop unswitching
      // - Value sinking
      // - Loop induction variable analysis:
      //   - Constant exit value folding
      //   - Bounds analysis
      //     - Constant-bound loop unrolling
      //     - Dead loop elimination (reducible loop with only
      //       pure actions and no external uses)
      //   - Strength reduction
      // - Instcombine
      //

      // 7. Rince and repeat
```

```
    if (ˆchgp) loop(src7) else src7}}
```

```
function genssa2_process(modenv, env, src) {
  ret = genssa2_process_iter(modenv, src);
  teq = genssa2_type_equations(modenv, env, ret);
  th = genssa2_solve_type_equations(modenv, env, teq);
  return th:ret;
}
```