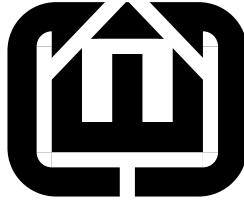


CEC GRC Optimizations



Cristian Soviani, Stephen A. Edwards
Columbia University
sedwards@cs.columbia.edu

Abstract

This cleans up useless nodes generated by the AST-to-GRC translation. Its centerpiece is a simple symbolic simulation algorithm that tracks the reachable states and exit levels from parallel threads to determine which code is dead.

Contents

1	Utility functions	2
1.1	find	2
1.2	erase	2
1.3	contains	2
1.4	delete_node	3
1.5	bypass	4
2	Symbolic Simulator	6
2.1	The Simulator	6
2.2	Switch	12
2.3	Enter	13
2.4	Terminate	13
2.5	Sync	15
2.6	Multiple Successors: Fork, Test	16
2.7	One successor: DefineSignal, Action, Nop, STSuspend	17
2.8	ExitGRC	18
2.9	DFS	19
2.10	ST Walk	19

3	Pass	20
3.1	DFS: forward and all	21
3.2	transform	22
4	STSimplify pass	23
5	RemoveNops	24
6	PruneSW	24
7	MergeSW Pass	25
8	DanglingST Pass	27
9	The main function	28

1 Utility functions

These are used by the various transforms to, e.g., search for elements in a vector and bypass control-flow graph nodes.

1.1 find

Return an iterator to the given object in the given vector.

2a *<General header 2a>* ≡ (29) 2b▷

```

template <class T> typename vector<T*>::iterator find(vector<T*> &v, T *e)
{
    for (typename vector<T*>::iterator i = v.begin() ; i != v.end() ; i++)
        if ( (*i) == e ) return i;
    return v.end();
}
```

1.2 erase

Erase the given object from the given vector.

2b *<General header 2a>* +≡ (29) <2a 2c>

```

template <class T> void erase(vector<T*> &v, T *e) {
    v.erase(find(v, e));
}
```

1.3 contains

Return true if the set contains the object.

2c *<General header 2a>* +≡ (29) <2b 3a>

```

template <class T> bool contains(set<T> &s, T o) {
    return s.find(o) != s.end();
}
```

3a \langle General header 2a $\rangle + \equiv$ (29) \langle 2c 3b \rangle

```

    template <class T, class U> bool contains(map<T, U> &m, T o) {
        return m.find(o) != m.end();
    }

```

1.4 delete_node

Delete a control-flow graph node and all references to it.

3b \langle General header 2a $\rangle + \equiv$ (29) \langle 3a 3d \rangle

```

    void delete_node(GRCNode *);

```

3c \langle GeneralUtils 3c $\rangle \equiv$ (30) 4a \rangle

```

    void delete_node(GRCNode *n)
    {
        assert(n);
        for ( vector<GRCNode*>::const_iterator i = n->successors.begin() ;
              i != n->successors.end() ; i++ )
            if (*i) erase((*i)->predecessors, n);
        for ( vector<GRCNode*>::const_iterator i = n->predecessors.begin() ;
              i != n->predecessors.end() ; i++ )
            *(find((*i)->successors, n)) = 0;
        for ( vector<GRCNode*>::const_iterator i = n->dataSuccessors.begin() ;
              i != n->dataSuccessors.end() ; i++ )
            erase((*i)->dataPredecessors, n);
        for ( vector<GRCNode*>::const_iterator i = n->dataPredecessors.begin() ;
              i != n->dataPredecessors.end() ; i++ )
            erase((*i)->dataSuccessors, n);

        // std::cerr << "Deleting a " << n->className() << std::endl;
        delete n;
    }

```

Delete a selection tree node.

3d \langle General header 2a $\rangle + \equiv$ (29) \langle 3b 4b \rangle

```

    void delete_node(STNode *);

```

4a $\langle \text{GeneralUtils } 3c \rangle + \equiv$ (30) $\langle 3c \ 5a \rangle$

```

void delete_node(STNode *n)
{
    assert(n);

    if (n->parent) {
        erase(n->parent->children, n);
        n->parent = 0;
    }

    for ( vector<STNode*>::const_iterator i = n->children.begin() ;
          i != n->children.end() ; i++ )
        (*i)->parent = 0;

    n->children.clear();

    delete n;
}

```

1.5 bypass

Bypass the given control-flow graph node: given a node with a single successor, point all its predecessors to that successor.

4b $\langle \text{General header } 2a \rangle + \equiv$ (29) $\langle 3d \ 5b \rangle$

```

void bypass(GRCNode *);

```

5a $\langle \text{GeneralUtils } 3c \rangle + \equiv$ (30) $\langle 4a \ 5c \rangle$

```

void bypass(GRCNode *n)
{
    assert(n);
    assert(n->successors.size() == 1);

    GRCNode *successor = n->successors.front();
    assert(successor);

    erase(successor->predecessors, n);
    n->successors.clear();

    for (vector<GRCNode*>::iterator i = n->predecessors.begin() ;
         i != n->predecessors.end() ; i++) {
        vector<GRCNode*>::iterator ip = find( (*i)->successors, n );
        assert(ip != (*i)->successors.end());
        (*ip) = successor;
        successor->predecessors.push_back(*i);
    }

    n->predecessors.clear();

    delete_node(n);
}

```

Bypass a selection-tree node: given a node with a single child, point its parent to that child.

5b $\langle \text{General header } 2a \rangle + \equiv$ (29) $\langle 4b \rangle$

```

void bypass(STNode *);

```

5c $\langle \text{GeneralUtils } 3c \rangle + \equiv$ (30) $\langle 5a \rangle$

```

void bypass(STNode *n)
{
    assert(n);
    assert(n->children.size() == 1);

    STNode *child = n->children.front();
    assert(child);

    assert(n->parent);
    vector<STNode*>::iterator i = find(n->parent->children, n);
    assert(i != n->parent->children.end());
    *i = child;

    child->parent = n->parent;

    n->parent = NULL;
    n->children.clear();
    // delete n; // FIXME: This should work, but it causes problems
}

```

2 Symbolic Simulator

Based on constructive semantics, this pass computes a conservative approximation of the reachable states (i.e., the children of each switch that can ever be active) and the exit levels attainable at each sync node.

It works by visiting nodes in the CFG that are reachable based on the states and termination levels reached. The `pending` set contains nodes scheduled to be analyzed. The main simulation loop consists of choosing a node from this set, marking it as visited, and visiting it, which usually involves scheduling some or all of its children. Enter nodes handle the state behavior. In addition to scheduling its child, an Enter node also re-schedules the switch to which it refers.

Sync nodes are the most complicated: they compute the set of termination levels that could be reached and schedule some or all of their children.

```

6a  <Simulator header 6a>≡ (29)
      class Simulator : public Visitor {
          GRCgraph &g;

          // Reached nodes in the control-flow graph and selection tree
          std::set<GRCNode *> cfg;
          std::set<STNode *> st;

          // Switch statement in the CFG for each exclusive node in the ST
          std::map<STexcl *, Switch *> switch_for_excl;

          // The set of known-reachable children of each sync node
          std::map<Sync *, set<int> > sync_levels;

          // Nodes scheduled to be analyzed
          std::set<GRCNode *> pending;

          GRCNode *entergrc;

          // All cfg and ST nodes
          std::set<GRCNode *> allcfg;
          std::set<STNode *> allst;

      public:
          <Simulator methods 6b>
          virtual ~Simulator() {}
      };

```

2.1 The Simulator

All the action happens here, in the constructor.

```

6b  <Simulator methods 6b>≡ (6a) 12b>
      Simulator(GRCgraph &);

```

The simulator operates in five phases: first, the initial conditions are set, then the main simulation loop is performed until there are no more nodes to visit, then dead successors of switch and exclusive nodes are removed, other dead CFG nodes are removed, and finally dead ST nodes are removed.

7a $\langle \text{Simulator body 7a} \rangle \equiv$ (30) 12a▷

```

    Simulator::Simulator(GRCgraph &gg) : g(gg)
    {
         $\langle \text{initialize simulator 7b} \rangle$ 
         $\langle \text{run simulator 8} \rangle$ 
         $\langle \text{fix switch nodes 9} \rangle$ 
         $\langle \text{fix sync nodes 10a} \rangle$ 
         $\langle \text{find and remove dead CFG nodes 10b} \rangle$ 
         $\langle \text{find and remove dead ST nodes 11} \rangle$ 
    }

```

Initialization: mark the root of the selection tree, its initial child, and the first node in the control-flow graph as reachable, and start the simulator at this node.

7b $\langle \text{initialize simulator 7b} \rangle \equiv$ (7a)

```

    STNode *stroot = g.selection_tree;
    assert(stroot);
    st.insert(stroot);

    // The root should have three children: initial, running, and terminated
    assert(stroot->children.size() == 3);
    st.insert(stroot->children[2]); // Mark initial child as reachable

    // Mark the topmost node in the control-flow graph (after the EnterGRC)
    // as reachable and mark it as needing to be visited

    assert(g.control_flow_graph);
    assert(g.control_flow_graph->successors.size() == 2);
    GRCNode *root = g.control_flow_graph->successors.back();
    assert(root);
    pending.insert(root);

```

The main simulation loop: remove a node from the pending set, mark it as visited, and simulate it. Note that certain nodes, specifically switch and sync, may be visited multiple times during the course of simulation. Others should be visited at most once.

```

8  <run simulator 8>≡ (7a)
    while ( !pending.empty() ) {
        set<GRCNode *>::iterator pi = pending.begin();
        GRCNode *n = *pi;
        assert(n);
        pending.erase(pi);
        // std::cerr << "Pending node " << cfgmap[n] << '\n';

        cfg.insert(n);
        n->welcome(*this);
    }

```


This code removes dead successors from switch nodes and dead children from the corresponding STexcl nodes. It maintains the number and order of successors and children under the two types of nodes.

This marches in lockstep two iterators through the successors of a switch and the children of the corresponding STexcl and removing unreached children. Because there may be additional fanin on nodes in the CFG, this algorithm uses the set of children reached in the selection tree to decide which children/successors are live.

```

9  <fix switch nodes 9>≡ (7a)
    for ( map<STexcl *, Switch *>::iterator i = switch_for_excl.begin() ;
          i != switch_for_excl.end() ; i++ ) {
        STexcl *excl = (*i).first;
        Switch *sw = (*i).second;
        /* std::cerr << "cleaning exclusive " << stmap[excl] << " and switch "
           << cfgmap[sw] << std::endl; */
        assert(excl);
        assert(sw);
        assert(excl->children.size() == sw->successors.size());

        // Remove the children/successors corresponding to unvisited
        // children in the Selection Tree
        vector<GRNode*>::iterator j = sw->successors.begin();
        vector<STNode*>::iterator k = excl->children.begin();
        while ( j != sw->successors.end() ) {
            assert(*j);
            assert(*k);
            if ( !contains(st, *k) ) {
                erase((*j)->predecessors, (GRNode*) sw);
                j = sw->successors.erase(j);
                (*k)->parent = 0;
                k = excl->children.erase(k);
            } else {
                j++;
                k++;
            }
        }
    }
}

```

This code NULLs out all successors that correspond to unreachable termination levels of each sync node. It uses the `sync_levels` map set up by the simulation rule for the Sync node.

```
10a  <fix sync nodes 10a>≡ (7a)
      for ( map<Sync *, set<int> >::iterator i = sync_levels.begin() ;
            i != sync_levels.end() ; i++ ) {
          Sync *sync = (*i).first;
          set<int> &levels = (*i).second;
          assert(sync);
          assert(!levels.empty());

          for ( vector<GRCNode*>::iterator j = sync->successors.begin() ;
                j != sync->successors.end() ; j++ ) {
            if ( *j && !contains(levels, j - sync->successors.begin()) ) {
              erase((*j)->predecessors, (GRCNode*) sync);
              *j = NULL;
            }
          }
      }
```

This code computes the set of all CFG nodes to be deleted by doing a comprehensive DFS of the CFG, then removes all the reached nodes from the visited set, and finally deletes each node in the resulting set.

```
10b  <find and remove dead CFG nodes 10b>≡ (7a)
      entergrc = g.control_flow_graph;
      assert(entergrc);
      assert(entergrc->successors.size() == 2);
      GRCNode *grcroot = entergrc->successors[1];
      all_dfs(grcroot);

      set<GRCNode *> unreachablecfg;
      set_difference( allcfg.begin(), allcfg.end(),
                     cfg.begin(), cfg.end(),
                     inserter(unreachablecfg, unreachablecfg.begin()) );

      // std::cerr << "Unreachable CFG nodes:";
      for ( set<GRCNode*>::const_iterator i = unreachablecfg.begin() ;
            i != unreachablecfg.end() ; i++ ) {
          // std::cerr << ' ' << cfgmap[*i];
          delete_node(*i);
      }
      // std::cerr << std::endl;
```

This code works similarly: it walks the existing selection tree and records all the nodes in it, removes the reachable nodes from this set, and finally deletes each node in the resulting set.

```

11  (find and remove dead ST nodes 11)≡ (7a)
    st_walk(stroot);

    set<STNode *> unreachablest;
    set_difference( allst.begin(), allst.end(),
                   st.begin(), st.end(),
                   inserter(unreachablest, unreachablest.begin()) );

    // std::cerr << "Unreachable ST nodes:";
    for ( set<STNode*>::const_iterator i = unreachablest.begin() ;
          i != unreachablest.end() ; i++ ) {
        // std::cerr << ' ' << stmap[*i];
        delete_node(*i);
    }
    // std::cerr << std::endl;

```

2.2 Switch

This schedules CFG nodes under the switch if the corresponding children of the corresponding STexcl in the ST have been entered.

```

12a  <Simulator body 7a>+≡ (30) <7a 13a>
      Status Simulator::visit(Switch &s)
      {
        // Remember our exclusive node for when we encounter Enter statements
        STexcl *excl = dynamic_cast<STexcl*>(s.st);
        assert(excl);
        switch_for_excl[excl] = &s;

        // Keep our STexcl node
        st.insert(excl);

        // Should always have the same number of children under the switch as
        // under the exclusive node in the ST
        assert(excl->children.size() == s.successors.size());

        // For each visited ST node under the exclusive, schedule the
        // corresponding successor of the switch if it hasn't already been visited

        for ( vector<STNode *>::const_iterator i = excl->children.begin() ;
              i != excl->children.end() ; i++ ) {
          if (contains(st, *i)) {
            // Determine the matching successor of this child
            GRCNode *suc = s.successors[i - excl->children.begin()];
            assert(suc); // All switch successors should be non-NULL
            // If the successor hasn't been visited already, schedule it
            if (!contains(cfg, suc)) pending.insert(suc);
          }
        }

        return Status();
      }

12b  <Simulator methods 6b>+≡ (6a) <6b 13b>
      Status visit(Switch &);

```

2.3 Enter

13a $\langle \text{Simulator body 7a} \rangle + \equiv$ (30) $\langle 12a \ 13c \rangle$

```

Status Simulator::visit(Enter &e)
{
    STNode *stnode = e.st;
    assert(stnode);

    // Find the STexcl node corresponding to our ST node
    // by climbing the tree until we hit one
    do {
        // Mark our ST node and all of our parents as visited
        st.insert(stnode);
        stnode = stnode->parent;
        assert(stnode); // Shouldn't go past the root, which should be an STexcl
    } while ( !dynamic_cast<STexcl*>(stnode) );
    STexcl *excl = dynamic_cast<STexcl*>(stnode);
    assert(excl); // Should have found the exclusive node

    // If we know about the corresponding switch node, schedule it
    if (contains(switch_for_excl, excl)) {
        assert(switch_for_excl[excl]);
        pending.insert(switch_for_excl[excl]);
    }

    // Schedule our successor
    assert(e.successors.size() == 1);
    assert(e.successors.front());
    pending.insert(e.successors.front());
    return Status();
}

```

13b $\langle \text{Simulator methods 6b} \rangle + \equiv$ (6a) $\langle 12b \ 14 \rangle$

```

Status visit(Enter &);

```

2.4 Terminate

Schedule our successor: a sync node. It will be scheduled multiple times, at most once per incoming Terminate, but that is fine.

13c $\langle \text{Simulator body 7a} \rangle + \equiv$ (30) $\langle 13a \ 15 \rangle$

```

Status Simulator::visit(Terminate &t)
{
    // Schedule our successor
    assert(t.successors.size() == 1);
    assert(t.successors.front());
    pending.insert(t.successors.front());
    return Status();
}

```

14 $\langle \textit{Simulator methods 6b} \rangle + \equiv$
 Status visit(Terminate &);

(6a) $\langle 13b \ 16a \rangle$

2.5 Sync

This is tricky: look at all the incoming Terminate nodes, determine the set of attainable exit levels, then schedule the corresponding successors if they have not already been scheduled.

The set of attainable exit levels is computed using the rule in Berry's *Constructive Semantics of Esterel*, i.e.,

$$\max(L, R) = \{i \mid i \geq \min(L)\} \cap (L \cup R) \cap \{j \mid j \geq \min(R)\}$$

That is, the union of all exit levels that are greater than the maximum over all minimum levels.

```

15  <Simulator body 7a>+≡ (30) <13c 16b>
    Status Simulator::visit(Sync &s)
    {
        // Count the number of threads coming into this sync: the maximum
        // index among all the preceeding terminate nodes

        int numThreads = 0;
        for ( vector<GRCCNode*>::const_iterator i = s.predecessors.begin() ;
              i != s.predecessors.end() ; i++ ) {
            Terminate *term = dynamic_cast<Terminate*>(*i);
            assert(term); // all predecessors should be Terminate nodes
            if (term->index >= numThreads) numThreads = term->index + 1;
        }

        vector<set<int> > levels(numThreads);

        // Build the sets of all exit levels by considering only those that
        // have been visited

        for ( vector<GRCCNode*>::const_iterator i = s.predecessors.begin() ;
              i != s.predecessors.end() ; i++ ) {
            Terminate *term = dynamic_cast<Terminate*>(*i);
            assert(term); // all predecessors should be Terminate nodes
            if (contains(cfg, (GRCCNode*) term))
                levels[term->index].insert(term->code);
        }

        int overallmin = 0;

        for ( vector<set<int> >::const_iterator i = levels.begin() ;
              i != levels.end() ; i++ ) {
            if ((*i).empty()) {
                // Don't know anything about one of the threads: need more
                // information before concluding which children may run,
                // so we'll stop here
                return Status();
            }
            int min = *((*i).begin());

```

```

    for ( set<int>::const_iterator j = (*i).begin() ;
          j != (*i).end() ; j++ )
        if ( (*j) < min ) min = *j;
    if (min > overallmin) overallmin = min;
}

// Compute the union of all levels greater or equal to the
// maximum of all the minimums

set<int> &level = sync_levels[&s];
level.clear();
for ( vector<set<int> >::const_iterator i = levels.begin() ;
      i != levels.end() ; i++ )
    for ( set<int>::const_iterator j = (*i).begin() ;
          j != (*i).end() ; j++ )
        if ( (*j) >= overallmin ) level.insert(*j);

// Schedule all the active children that aren't already visited

for ( vector<GRNode*>::const_iterator i = s.successors.begin() ;
      i != s.successors.end() ; i++ )
    if ( contains(level, (int) (i - s.successors.begin())) &&
        !contains(cfg, *i) ) {
        assert(*i);
        pending.insert(*i);
    }

return Status();
}

```

16a $\langle \text{Simulator methods } 6b \rangle + \equiv$
 Status visit(Sync &);

(6a) $\langle 14 \ 16c \rangle$

2.6 Multiple Successors: Fork, Test

Schedule all our successors.

16b $\langle \text{Simulator body } 7a \rangle + \equiv$
 Status Simulator::visit(Fork &f)
 {
 for (vector<GRNode*>::const_iterator i = f.successors.begin() ;
 i != f.successors.end() ; i++) {
 assert(*i);
 pending.insert(*i);
 }
 return Status();
 }

(30) $\langle 15 \ 17a \rangle$

16c $\langle \text{Simulator methods } 6b \rangle + \equiv$
 Status visit(Fork &);

(6a) $\langle 16a \ 17b \rangle$

17a $\langle \text{Simulator body 7a} \rangle + \equiv$ (30) $\triangleleft 16b \ 17c \triangleright$

```

Status Simulator::visit(Test &s)
{
    for (vector<GRNode*>::const_iterator i = s.successors.begin() ;
        i != s.successors.end() ; i++ ) {
        assert(*i);
        pending.insert(*i);
    }
    return Status();
}

```

17b $\langle \text{Simulator methods 6b} \rangle + \equiv$ (6a) $\triangleleft 16c \ 17d \triangleright$

```

Status visit(Test &);

```

2.7 One successor: DefineSignal, Action, Nop, STSuspend

17c $\langle \text{Simulator body 7a} \rangle + \equiv$ (30) $\triangleleft 17a \ 17e \triangleright$

```

Status Simulator::visit(DefineSignal &d)
{
    // Schedule our successor
    assert(d.successors.size() == 1);
    assert(d.successors.front());
    pending.insert(d.successors.front());
    return Status();
}

```

17d $\langle \text{Simulator methods 6b} \rangle + \equiv$ (6a) $\triangleleft 17b \ 17f \triangleright$

```

Status visit(DefineSignal &);

```

17e $\langle \text{Simulator body 7a} \rangle + \equiv$ (30) $\triangleleft 17c \ 18a \triangleright$

```

Status Simulator::visit(Action &a)
{
    // Schedule our successor
    assert(a.successors.size() == 1);
    assert(a.successors.front());
    pending.insert(a.successors.front());
    return Status();
}

```

17f $\langle \text{Simulator methods 6b} \rangle + \equiv$ (6a) $\triangleleft 17d \ 18b \triangleright$

```

Status visit(Action &);

```

18a $\langle \text{Simulator body 7a} \rangle + \equiv$ (30) $\langle 17e \ 18c \rangle$
 Status Simulator::visit(Nop &s)
 {
 // Schedule our successor
 assert(s.successors.size() == 1);
 assert(s.successors.front());
 pending.insert(s.successors.front());
 return Status();
 }

 18b $\langle \text{Simulator methods 6b} \rangle + \equiv$ (6a) $\langle 17f \ 18d \rangle$
 Status visit(Nop &);

 18c $\langle \text{Simulator body 7a} \rangle + \equiv$ (30) $\langle 18a \ 19a \rangle$
 Status Simulator::visit(STSuspend &s)
 {
 // Schedule our successor
 assert(s.successors.size() == 1);
 assert(s.successors.front());
 pending.insert(s.successors.front());
 return Status();
 }

 18d $\langle \text{Simulator methods 6b} \rangle + \equiv$ (6a) $\langle 18b \ 18e \rangle$
 Status visit(STSuspend &);

2.8 ExitGRC

Do nothing: we have reached the end.

18e $\langle \text{Simulator methods 6b} \rangle + \equiv$ (6a) $\langle 18d \ 19b \rangle$
 Status visit(ExitGRC &) { return Status(); }

2.9 DFS

Visit all the nodes in the control-flow graph and record them in the `allcfg` set.

```

19a  <Simulator body 7a>+≡ (30) <18c 19c>
      void Simulator::all_dfs(GRCNode *n)
      {
        if ( !n || n == entergrc || contains(allcfg, n) ) return;

        allcfg.insert(n);

        for (vector<GRCNode*>::const_iterator ch = n->successors.begin();
              ch != n->successors.end(); ch++) all_dfs(*ch);
        for (vector<GRCNode*>::const_iterator ch = n->predecessors.begin();
              ch != n->predecessors.end(); ch++) all_dfs(*ch);
        for (vector<GRCNode*>::const_iterator ch = n->dataSuccessors.begin();
              ch != n->dataSuccessors.end(); ch++) all_dfs(*ch);
        for (vector<GRCNode*>::const_iterator ch = n->dataPredecessors.begin();
              ch != n->dataPredecessors.end(); ch++) all_dfs(*ch);
      }

19b  <Simulator methods 6b>+≡ (6a) <18e 19d>
      void all_dfs(GRCNode *);

```

2.10 ST Walk

Visit all the nodes in the selection tree and record them in the `allst` set.

```

19c  <Simulator body 7a>+≡ (30) <19a
      void Simulator::st_walk(STNode *n)
      {
        allst.insert(n);

        for (vector<STNode*>::const_iterator ch = n->children.begin();
              ch != n->children.end(); ch++)
          st_walk(*ch);
      }

19d  <Simulator methods 6b>+≡ (6a) <19b
      void st_walk(STNode *);

```

3 Pass

This class deletes unreachable nodes then visits all reachable control-flow graph nodes. The visit methods do nothing by default, but those in classes derived from this one do modify the nodes.

```

20  <Pass header 20>≡ (29)
    class Pass : public Visitor {
        Status visit(Switch &) { return Status(); }
        Status visit(Test &) { return Status(); }
        Status visit(Terminate &) { return Status(); }
        Status visit(Sync &) { return Status(); }
        Status visit(Fork &) { return Status(); }
        Status visit(Action &) { return Status(); }
        Status visit(Enter &) { return Status(); }
        Status visit(STSuspend &) { return Status(); }
        Status visit(DefineSignal &) { return Status(); }
        Status visit(Nop &) { return Status(); }

        std::vector<GRCNode*> toplist;
        std::set<GRCNode*> reachable_nodes;
        std::set<GRCNode*> all_nodes;
        bool forward;
        GRCNode *entergrc;
        GRCNode *exitgrc;
        GRCNode *grcroot;
        STNode *stroot;

        <Pass declarations 21a>

    public:
        Pass(GRCgraph* g, bool f) : forward(f) {
            assert(g);
            stroot = g->selection_tree;
            assert(stroot);
            entergrc = g->control_flow_graph;
            assert(entergrc);
            assert(entergrc->successors.size() == 2);
            exitgrc = entergrc->successors[0];
            grcroot = entergrc->successors[1];
        }
        virtual ~Pass(){}

        void transform();
    };

```

3.1 DFS: forward and all

Perform a depth-first search of the control-flow graph nodes. Only forward control dependencies are considered, so unreachable nodes are truly unreachable.

21a \langle Pass declarations 21a $\rangle \equiv$ (20) 21c \triangleright
`void forward_dfs(GRCNode *);`

21b \langle Pass body 21b $\rangle \equiv$ (30) 21d \triangleright
`void Pass::forward_dfs(GRCNode *n)`
`{`
 `if (!n || n == exitgrc || contains(reachable_nodes, n)) return;`

 `reachable_nodes.insert(n);`

 `for(vector<GRCNode*>::const_iterator ch = n->successors.begin();`
 `ch != n->successors.end(); ch++) forward_dfs(*ch);`

 `toplist.push_back(n);`
`}`

Perform a depth-first search of the control-flow graph. Traverse both control and data predecessors and successors. This visits all nodes that any node knows about.

21c \langle Pass declarations 21a $\rangle + \equiv$ (20) \triangleleft 21a
`void all_dfs(GRCNode *);`

21d \langle Pass body 21b $\rangle + \equiv$ (30) \triangleleft 21b 22 \triangleright
`void Pass::all_dfs(GRCNode *n)`
`{`
 `if (!n || n == exitgrc || n == entergrc || contains(all_nodes, n)) return;`

 `all_nodes.insert(n);`

 `for (vector<GRCNode*>::const_iterator ch = n->successors.begin();`
 `ch != n->successors.end(); ch++) all_dfs(*ch);`
 `for (vector<GRCNode*>::const_iterator ch = n->predecessors.begin();`
 `ch != n->predecessors.end(); ch++) all_dfs(*ch);`
 `for (vector<GRCNode*>::const_iterator ch = n->dataSuccessors.begin();`
 `ch != n->dataSuccessors.end(); ch++) all_dfs(*ch);`
 `for (vector<GRCNode*>::const_iterator ch = n->dataPredecessors.begin();`
 `ch != n->dataPredecessors.end(); ch++) all_dfs(*ch);`
`}`

3.2 transform

This is the main entry point for a pass: it removes all unreachable nodes then visits all others in either forward or reverse topological order.

To do this, it performs two depth-first searches to identify all the nodes reachable through forward control dependence (i.e., all that could ever possibly run), then all reachable through any dependence, remove those from the second search that were not found in the first search (i.e., all the unreachable nodes), then either visits each reachable node in either forward or reverse topological order, depending on the setting of the **forward** flag passed when the pass was constructed.

```

22  <Pass body 21b>+= (30) <21d
    void Pass::transform()
    {
        forward_dfs(grcroot); // build toplist
        all_dfs(grcroot);

        set<GRCNode *> unreachable;
        set_difference( all_nodes.begin(), all_nodes.end(),
                        reachable_nodes.begin(), reachable_nodes.end(),
                        inserter(unreachable, unreachable.begin()) );

        for (set<GRCNode *>::const_iterator i = unreachable.begin();
             i != unreachable.end() ; i++)
            delete_node(*i);

        if (forward)
            for (vector<GRCNode *>::iterator i = toplist.begin() ;
                 i != toplist.end() ; i++ ) (*i)->welcome(*this);
        else
            for (vector<GRCNode *>::reverse_iterator i = toplist.rbegin() ;
                 i != toplist.rend() ; i++ ) (*i)->welcome(*this);
    }

```

4 STSimplify pass

Removes all needless nodes from the ST (mostly ref nodes).

FIXME: This should be replaced with something derived from the Visitor class to avoid all the dynamic_casts.

```

23a  <STSimplify header 23a>≡ (29)
      class STSimplify {
          GRCgraph *g;
          std::set<STNode*> &stkept;

          STNode *check_st(STNode *, STNode *realpar);
      public:
          STSimplify(GRCgraph *g, std::set<STNode*> &stkept)
              : g(g), stkept(stkept) { assert(g); }
          void simplify() { g->selection_tree = check_st(g->selection_tree, NULL); }
      };

23b  <STSimplify body 23b>≡ (30)
      STNode *STSimplify::check_st(STNode *n, STNode *realpar)
      {
          bool keep = false;
          STNode* c;
          STref *ref;
          int is_simpleref;

          n->parent = realpar;

          if (dynamic_cast<STleaf*>(n) ) {
              stkept.insert(n);
              return n;
          }

          is_simpleref = 0;
          if((ref=dynamic_cast<STref*>(n)))
              is_simpleref = ! (ref->isabort() || ref->issuspend());

          if(!is_simpleref) realpar = n; // try to keep

          c=NULL; keep = false;
          for(vector<STNode*>::iterator i=n->children.begin(); i!=n->children.end(); i++)
              if(*i){
                  (*i) = c = check_st(*i, realpar);
                  if(c) keep=1;
              }

          if(is_simpleref)
              if(keep) return c; else return NULL;

          if(keep) { stkept.insert(n); return n; } else return NULL;
      }

```

5 RemoveNops

Remove the Defs nodes. This step can be enabled or not, we are not decided yet

24a $\langle \text{RemoveNops header 24a} \rangle \equiv$ (29)

```
class RemoveNops : public Pass {
    Status visit(DefineSignal &);
    Status visit(Nop &);
public:
    RemoveNops(GRCgraph *g) : Pass(g, true) {};
};
```

24b $\langle \text{RemoveNops body 24b} \rangle \equiv$ (30)

```
Status RemoveNops::visit(DefineSignal &s){
    bypass(&s);
    return Status();
}
Status RemoveNops::visit(Nop &s){
    bypass(&s);
    return Status();
};
```

6 PruneSW

For each switch statement, remove every child corresponding to a removed selection tree node.

24c $\langle \text{PruneSW header 24c} \rangle \equiv$ (29)

```
class PruneSW : public Pass {
    std::set<STNode*> &stkept;
    Status visit(Switch &);
public:
    PruneSW(GRCgraph* g, std::set<STNode*> &stkept) :
        Pass(g, false), stkept(stkept) {}
};
```


25a *(PruneSW body 25a)*≡ (30)

```

Status PruneSW::visit(Switch &s)
{
    for ( vector<STNode*>::iterator sch = s.st->children.begin() ;
          sch != s.st->children.end() ; ) {
        if (*sch && contains(stkept, *sch) ) {
            sch++;
        } else {
            // The decision was made to delete the selection tree node
            // corresponding to this child of the switch.
            // Remove the arc from the switch to the corresponding child
            vector<GRCNode*>::iterator ch =
                s.successors.begin() + (sch - s.st->children.begin());
            erase( (*ch)->predecessors, (GRCNode*) &s );
            s.successors.erase(ch);
            sch = s.st->children.erase(sch); // now sch points to the next element
        }
    }

    // remove switches with only 1 child
    if ( s.successors.size() == 1 ) {
        bypass(&s); // Remove the switch

        // Each switch in the ST should have exactly one switch
        // in the control-flow graph, so we should never encounter
        // an already-removed switch
        assert( contains(stkept, s.st) );

        bypass(s.st); // Remove the selection tree node
        stkept.erase(s.st); // Mark the selection tree node as gone
    }

    return Status();
}

```

7 MergeSW Pass

Merge cascaded switches. All the children are inserted in the place of the child switch. The same operations are done (carefully) in the selection tree.

25b *(MergeSW header 25b)*≡ (29)

```

class MergeSW : public Pass {
    std::set<STNode*> &stkept;
    Status visit(Switch &);
public:
    MergeSW(GRCgraph* g, std::set<STNode*> &stkept) :
        Pass(g, false), stkept(stkept) {}
};

```

26 $\langle \text{MergeSW body 26} \rangle \equiv$ (30)

```

Status MergeSW::visit(Switch &s)
{
    if (s.predecessors.size() > 1)
        return Status();

    int szc = s.successors.size();
    assert(szc == (int) s.st->children.size() );

    Switch *p = dynamic_cast<Switch*>(s.predecessors.front());
    if (!p) return Status();

    // Parent of this switch is also a switch: merge

    vector<GRCNode*>::iterator ip = find(p->successors, (GRCNode*) &s);
    int chno = ip-p->successors.begin();
    ip = p->successors.erase(ip);

    for (int ich = 0; ich < szc; ich++ ) {
        GRCNode *ch = s.successors[ich];
        vector<GRCNode*>::iterator i = find(ch->predecessors, (GRCNode*) &s);
        (*i)=p;
        ip = p->successors.insert(ip, ch);
        ip++;
    }

    STNode *st_par = p->st;
    vector<STNode*>::iterator ips = st_par->children.begin() + chno;
    ips = st_par->children.erase(ips);

    for (int ich = 0 ; ich < szc ; ich++) {
        STNode *chs = s.st->children[ich];
        chs->parent = st_par;
        ips = st_par->children.insert(ips, chs);
        ips++;
    }

    stkept.erase(s.st);

    return Status();
}

```

8 DanglingST Pass

If the ST node corresponding to a given Enter or STSuspend was removed, remove the Enter or STSuspend by bypassing it.

27a $\langle \text{DanglingST header 27a} \rangle \equiv$ (29)

```
class DanglingST : public Pass {
    std::set<STNode*> &stkept;
    Status visit(Enter &);
    Status visit(STSuspend &);
public:
    DanglingST(GRCgraph* g, std::set<STNode*> &stkept) :
        Pass(g, true), stkept(stkept) {}
};
```

27b $\langle \text{DanglingST body 27b} \rangle \equiv$ (30)

```
Status DanglingST::visit(Enter &s)
{
    if ( !contains(stkept, s.st) ) {
        // std::cerr << "Dangling Enter: " << cfgmap[&s] << '\n';
        bypass(&s);
    }
    return Status();
}

Status DanglingST::visit(STSuspend &s)
{
    if ( !contains(stkept, s.st) ) {
        // std::cerr<<"Dangling STSuspend: "<<cfgmap[&s]<<'\n';
        bypass(&s);
    }
    return Status();
}
```

9 The main function

This is the main() function, which invokes the various optimization passes.

```

28  <main 28>≡ (30)
    int main(int argc, char *argv[])
    {
        try {
            IR::XMListream r(std::cin);
            IR::Node *n;
            r >> n;

            AST::Modules *mods = dynamic_cast<AST::Modules*>(n);
            if (!mods) throw IR::Error("Root node is not a Modules object");

            for ( std::vector<AST::Module*>::iterator i = mods->modules.begin() ;
                  i != mods->modules.end() ; i++ ) {
                AST::Module* mod = *i;
                assert(mod);

                AST::GRCgraph *g = dynamic_cast<AST::GRCgraph*>(mod->body);
                if (!g) throw IR::Error("Module is not in GRC format");

                g->enumerate(GRCOpt::cfgmap, GRCOpt::stmap);

                // Remove unreachable nodes by performing symbolic simulation
                GRCOpt::Simulator sim(*g);

                // Set that contains the selection tree nodes to be preserved
                std::set<AST::STNode*> stkept;

                // Remove needless ST nodes
                GRCOpt::STSimplify pass1(g, stkept);
                pass1.simplify();

                GRCOpt::RemoveNops pass1a(g);
                pass1a.transform();

                // Remove children of switches corresponding to needless ST nodes
                GRCOpt::PruneSW pass2(g, stkept);
                pass2.transform();

                // Merge cascaded switches
                GRCOpt::MergeSW pass3(g, stkept);
                pass3.transform();

                // Remove Enter and STSuspend nodes corresponding to removed ST nodes
                GRCOpt::DanglingST pass4(g, stkept);
                pass4.transform();

```

```

        // Remove any unreachable nodes
        GRCOpt::Pass pass5(g, true);
        pass5.transform();

    }

    // end of transformations

    IR::XMLostream w(std::cout);
    w << n;

    } catch (IR::Error &e) {
        std::cerr << e.s << std::endl;
        exit(-1);
    }
    return 0;
}

```

```

29  <GRCOpt.hpp 29>≡
    #include "IR.hpp"
    #include "AST.hpp"
    #include <set>
    #include <vector>

    namespace GRCOpt {

        using namespace AST;

        using std::set;
        using std::vector;

        <General header 2a>
        <Simulator header 6a>
        <Pass header 20>
        <DanglingST header 27a>
        <PruneSW header 24c>
        <MergeSW header 25b>
        <STSimplify header 23a>
        <RemoveNops header 24a>
    }

```

```
30  <GRCOpt.cpp 30>≡
    #include "GRCOpt.hpp"
    #include "GRCPrinter.hpp"
    #include <iostream>
    #include <algorithm>

    namespace GRCOpt {

        GRCNode::NumMap cfgmap;
        STNode::NumMap stmap;

        <GeneralUtils 3c>
        <Simulator body 7a>
        <Pass body 21b>
        <DanglingST body 27b>
        <PruneSW body 25a>
        <MergeSW body 26>
        <STSimplify body 23b>
        <RemoveNops body 24b>
    }

    <main 28>
```