CEC AST-to-GRC Translator

Stephen A. Edwards
Columbia University
sedwards@cs.columbia.edu

Contents

1 GRC Synthesis

1.1 The Context class

1 ⟨*context class* 1⟩≡ (26c)

```
struct Context {
  int size;
  std::stack<GRCNode**> continuations;

  Context(int max_code) : size(max_code) {
    continuations.push(new GRCNode*[size]);
    for (int i = 0 ; i < size ; i++ ) continuations.top()[i] = 0;
  }
  ~Context() {}
```

```
  void push(Context &c) {
    GRCNode **parent = c.continuations.top();
    continuations.push(new GRCNode*[size]);
    GRCNode **child = continuations.top();
    for ( int i = 0 ; i < size ; i++ ) child[i] = parent[i];
  }

  void push() { push(*this); }

  void pop() {
    delete [] continuations.top();
    continuations.pop();
  }

  GRCNode *& operator ()(int k) { return continuations.top()[k]; }
};
```

1.2 The GrcSynth class

2a ⟨*GrcSynth class* 2a⟩≡ (26c)

```
struct GrcSynth {
  Module *module;
  Context surface_context;
  Context depth_context;
  Context seltree_context;
  Surface surface;
  Depth depth;
  SelTree seltree;
  map<GRCNode *, STNode *> grc2st;

  BuiltinTypeSymbol *integer_type;
  BuiltinTypeSymbol *boolean_type;

  ⟨GrcSynth methods 2b⟩
};
```

2b ⟨*GrcSynth methods* 2b⟩≡ (2a) 3▷

```
GrcSynth(Module *m)
  : module(m),
    surface_context(m->max_code+1), depth_context(m->max_code+1), seltree_context(m->max_code+
    surface(surface_context, *this), depth(depth_context, *this), seltree(seltree_context, *th
{
  assert(m->types);
  integer_type = dynamic_cast<BuiltinTypeSymbol*>(m->types->get("integer"));
  assert(integer_type);
  boolean_type = dynamic_cast<BuiltinTypeSymbol*>(m->types->get("boolean"));
  assert(boolean_type);
}
```

3 ⟨*GrcSynth methods* 2b⟩+≡ (2a) ◁2b

```
GRCNode *synthesize()
{
  GRCNode *synt_seltree, *synt_surface, *synt_depth;

  TopGRC *top;
  Switch *top_switch;
  Leave *lv_flow, *lv_boot;
  STexcl *stroot;
  STleaf *boot, *finished;
  Enter *enfinished, *finalloop;
  Terminate *term0, *term1;

  assert(module->body);

   stroot=new STexcl();
   boot=new STleaf();
   finished=new STleaf();
   finished->setfinal();

   top=new TopGRC();
   lv_boot =new Leave(); lv_flow=new Leave();
   enfinished=new Enter();
   finalloop = new Enter();
   top_switch = new Switch();

   lv_boot->st=boot;
   top_switch->st=stroot;
   lv_flow->st=stroot;
   enfinished->st=finished;
   finalloop->st=finished;

   *lv_flow >> enfinished;
   term0=new Terminate(0); *term0 >> lv_flow;
   term1=new Terminate(1);
   surface_context(0) = depth_context(0) = seltree_context(0) = term0;
   surface_context(1) = depth_context(1) = term1;


   synt_seltree=seltree.synthesize(module->body);
   synt_surface=surface.synthesize(module->body);
   synt_depth=depth.synthesize(module->body);

   *stroot >> finished >> synt_seltree >> boot;

   *lv_boot >> synt_surface;

   *top_switch >> finalloop >> synt_depth >> lv_boot;

   *top >> stroot >> top_switch;
```

```
    std::cerr<<"SYNT OK"<<'\n';

    return top;
}
```

1.3 The Surface and Depth classes

4a ⟨*grc walker class* 4a⟩≡ (26c)

```
class GrcWalker : public Visitor {
protected:
  Context &context;
  GrcSynth &environment;
public:
  GrcWalker(Context &c, GrcSynth &e) : context(c), environment(e) {};

  GRCNode *synthesize(ASTNode *n) {
    assert(n);
    n->welcome(*this);
    assert(context(0));
    return context(0);
  }

  GRCNode *recurse(ASTNode *n) {
    context.push();
    GRCNode *nn = synthesize(n);
    context.pop();
    return nn;
  }

  static GRCNode* push_onto(GRCNode *& b, GRCNode* n) {
    *n >> b;
    b = n;
    return b;
  }
};
```

4b ⟨*surface class* 4b⟩≡ (26c)

```
class Surface : public GrcWalker {
public:
  Surface(Context &c, GrcSynth &e) : GrcWalker(c, e) {}
  ⟨surface methods 5b⟩
};
```

4c ⟨*depth class* 4c⟩≡ (26c)

```
class Depth : public GrcWalker {
public:
  Depth(Context &c, GrcSynth &e) : GrcWalker(c, e) {}
  ⟨depth methods 5d⟩
};
```

5a ⟨*st class* 5a⟩≡ (26c)
```
class SelTree : public GrcWalker {
public:
  SelTree(Context &c, GrcSynth &e): GrcWalker(c,e) {}
  ⟨st methods 5f⟩
};
```

1.4 Pause

The surface of a pause Enters and terminates at level 1.

5b ⟨*surface methods* 5b⟩≡ (4b) 6b▷
```
Status visit(Pause &);
```

5c ⟨*surface method definitions* 5c⟩≡ (26d) 8a▷
```
Status Surface::visit(Pause &s) {
  Enter *en = new Enter();
  en->st=environment.grc2st[(GRCNode *)&s];
  std::cerr<<"Pause surf st="<<en->st<<std::endl;

  *en >> context(1);
  context(0) = en;
  return Status();
}
```

The depth of a pause is a Leave.

5d ⟨*depth methods* 5d⟩≡ (4c) 6c▷
```
Status visit(Pause &);
```

5e ⟨*depth method definitions* 5e⟩≡ (26d) 8c▷
```
Status Depth::visit(Pause &s) {
//  Leave *lv;
//  lv=new Leave();
//  lv->st=environment.grc2st[(GRCNode*)&s];
//  std::cerr<<"Pause depth st="<<lv->st<<std::endl;
  Switch *sw;
  sw = new Switch();
  sw -> st = (STNode*) (environment.grc2st[(GRCNode*)&s])->predecessors[0];
  push_onto(context(0), sw);

  return Status();
}
```

5f ⟨*st methods* 5f⟩≡ (5a) 6d▷
```
Status visit(Pause &);
```

6a ⟨*st method definitions* 6a⟩≡ (26d) 7e▷
```
Status SelTree::visit(Pause &s){
  STleaf *leaf;
  STexcl *excl;
  std::cerr<<"Pause st"<<std::endl;
  leaf=new STleaf();
  excl = new STexcl(); *excl >> leaf;
  environment.grc2st[(GRCNode*)&s] = leaf;
  context(0)= excl;
  return Status();
}
```

1.5 Exit

This sends the incoming activation to the code for the exit.

6b ⟨*surface methods* 5b⟩+≡ (4b) ◁5b 6e▷
```
Status visit(Exit &s) {
  assert(s.trap);
  assert(s.trap->code > 0);
  assert(context(s.trap->code));
  context(0) = context(s.trap->code);
  return Status();
}
```

6c ⟨*depth methods* 5d⟩+≡ (4c) ◁5d 6f▷
```
Status visit(Exit &) { return Status(); }
```

6d ⟨*st methods* 5f⟩+≡ (5a) ◁5f 7a▷
```
Status visit(Exit &) {
  context(0)=new STref();
  return Status();
}
```

1.6 Emit

This becomes an action in the surface; the depth is vacuous.

6e ⟨*surface methods* 5b⟩+≡ (4b) ◁6b 7b▷
```
Status visit(Emit &s) {
  Action *a = new Action(&s);
  *a >> context(0);
  context(0) = a;
  return Status();
}
```

6f ⟨*depth methods* 5d⟩+≡ (4c) ◁6c 7c▷
```
Status visit(Emit &) { return Status(); }
```

7a ⟨*st methods* 5f⟩+≡ (5a) ◁6d 7d▷

```
Status visit(Emit &) {
  context(0)=new STref();
  return Status();
}
```

1.7 Assignment

7b ⟨*surface methods* 5b⟩+≡ (4b) ◁6e 7f▷

```
Status visit(Assignment &s) {
  Action *a = new Action(&s);
  *a >> context(0);
  context(0) = a;
  return Status();
}
```

7c ⟨*depth methods* 5d⟩+≡ (4c) ◁6f 8b▷

```
Status visit(Assignment &) { return Status(); }
```

1.8 IfThenElse

7d ⟨*st methods* 5f⟩+≡ (5a) ◁7a 8f▷

```
Status visit(IfThenElse &);
```

7e ⟨*st method definitions* 6a⟩+≡ (26d) ◁6a 9a▷

```
Status SelTree::visit(IfThenElse &s) {
  STexcl *ite;
  ite=new STexcl();
  environment.grc2st[(GRCNode*)&s] = ite;

  *ite >> (s.else_part ? synthesize(s.else_part) : NULL);
  *ite >> (s.then_part ? synthesize(s.then_part) : NULL);

  context(0)= ite;
  return Status();
}
```

7f ⟨*surface methods* 5b⟩+≡ (4b) ◁7b 8d▷

```
Status visit(IfThenElse &);
```

8a ⟨*surface method definitions* 5c⟩+≡ (26d) ◁5c 9b▷

```
Status Surface::visit(IfThenElse &s) {
  Leave *lv;
  Enter *en;
  assert(s.predicate);
  lv = new Leave(); lv->st=environment.grc2st[(GRCNode*)&s];
  push_onto(context(0), lv);
  Test *t = new Test(s.predicate);
     t->st = environment.grc2st[(GRCNode*)&s];
  *t >> ( (s.else_part != 0) ? recurse(s.else_part) : context(0))
     >> ( (s.then_part != 0) ? recurse(s.then_part) : context(0));
  context(0) = t;
  en = new Enter(); en->st=environment.grc2st[(GRCNode*)&s];
  push_onto(context(0), en);
  return Status();
}
```

8b ⟨*depth methods* 5d⟩+≡ (4c) ◁7c 8e▷

```
Status visit(IfThenElse &);
```

8c ⟨*depth method definitions* 5e⟩+≡ (26d) ◁5e 10a▷

```
Status Depth::visit(IfThenElse &s) {
  Leave *lv;
  lv=new Leave(); lv->st=environment.grc2st[(GRCNode*)&s];
  push_onto(context(0), lv);
  Switch *sw = new Switch(); sw->st=environment.grc2st[(GRCNode*)&s];
  *sw >> ( (s.else_part != 0) ? recurse(s.else_part) : context(0))
      >> ( (s.then_part != 0) ? recurse(s.then_part) : context(0));
  context(0) = sw;
  return Status();
}
```

1.9 StatementList

Sequencing is slightly difficult because of need to handle reincarnation.

8d ⟨*surface methods* 5b⟩+≡ (4b) ◁7f 10b▷

```
Status visit(StatementList &);
```

8e ⟨*depth methods* 5d⟩+≡ (4c) ◁8b 10c▷

```
Status visit(StatementList &);
```

8f ⟨*st methods* 5f⟩+≡ (5a) ◁7d 10d▷

```
Status  visit(StatementList &s);
```

9a ⟨*st method definitions* 6a⟩+≡ (26d) ◁7e 10e▷

```
Status  SelTree::visit(StatementList &s)
{
  STexcl *excl;

  excl=new STexcl();
  environment.grc2st[(GRCNode*)&s] = excl;

  for ( vector<Statement*>::reverse_iterator i = s.statements.rbegin() ;
        i != s.statements.rend() ; i++ ){
    assert(*i);
    *excl >> synthesize(*i);
  }

  context(0)=excl;
  std::cerr<<"seq ok"<<std::endl;
  return Status();
}
```

9b ⟨*surface method definitions* 5c⟩+≡ (26d) ◁8a 11a▷

```
Status Surface::visit(StatementList &s) {
  Leave *lv;
  Enter *en;
  lv=new Leave();
  lv->st=environment.grc2st[(GRCNode*)&s];
  push_onto(context(0), lv);
  for ( vector<Statement*>::reverse_iterator i = s.statements.rbegin() ;
        i != s.statements.rend() ; i++ ) {
    assert(*i);
    context(0) = synthesize(*i);
  }
  en=new Enter();
  en->st=environment.grc2st[(GRCNode*)&s];
  push_onto(context(0), en);
  return Status();
}
```

10a ⟨*depth method definitions* 5e⟩+≡ (26d) ◁8c 11b▷

```
Status Depth::visit(StatementList &s) {
  Leave *lv;
  Switch *sw;
  if (!s.statements.empty()) {
    lv=new Leave();
    lv->st=environment.grc2st[(GRCNode*)&s];
    push_onto(context(0), lv);
    sw = new Switch();
    sw->st=environment.grc2st[(GRCNode*)&s];
    environment.surface_context.push(context);
    vector<Statement*>::reverse_iterator final = s.statements.rend();
    final--;
    for ( vector<Statement*>::reverse_iterator i = s.statements.rbegin() ;
          i != s.statements.rend() ; i++ ) {
      assert(*i);
      *sw >> synthesize(*i); // Build the depth
      // Build the surface
      if (i != final ) context(0) = environment.surface.synthesize(*i);
    }
    environment.surface_context.pop();
    context(0) = sw;
  }
  return Status();
}
```

1.10 Loop

Loops duplicate their surface surface.

10b ⟨*surface methods* 5b⟩+≡ (4b) ◁8d 11c▷

```
Status visit(Loop &);
```

10c ⟨*depth methods* 5d⟩+≡ (4c) ◁8e 11d▷

```
Status visit(Loop &);
```

10d ⟨*st methods* 5f⟩+≡ (5a) ◁8f 11e▷

```
Status visit(Loop &s);
```

10e ⟨*st method definitions* 6a⟩+≡ (26d) ◁9a 14a▷

```
Status SelTree::visit(Loop &s){
  STref *lp;
  lp=new STref();
  environment.grc2st[(GRCNode*)&s] = lp;
  *lp>>synthesize(s.body);
  context(0)=lp;
  return Status();
}
```

11a ⟨*surface method definitions* 5c⟩+≡ (26d) ◁9b 12▷

```
Status Surface::visit(Loop &s) {
  Enter *en;
  context(0) = synthesize(s.body);
  en=new Enter(); en->st=environment.grc2st[(GRCNode*)&s];
  push_onto(context(0), en);
  return Status();
}
```

11b ⟨*depth method definitions* 5e⟩+≡ (26d) ◁10a 13▷

```
Status Depth::visit(Loop &s) {
  environment.surface_context.push(context);
  // Synthesize surface
  context(0) = environment.surface.synthesize(s.body);
  // Synthesize depth
  context(0) = synthesize(s.body);
  environment.surface_context.pop();
  return Status();
}
```

1.11 Suspend

11c ⟨*surface methods* 5b⟩+≡ (4b) ◁10b 14b▷

```
Status visit(Suspend &);
```

11d ⟨*depth methods* 5d⟩+≡ (4c) ◁10c 14c▷

```
Status visit(Suspend &);
```

11e ⟨*st methods* 5f⟩+≡ (5a) ◁10d 14d▷

```
Status visit(Suspend &);
```

12 ⟨*surface method definitions* 5c⟩+≡ (26d) ◁11a 15▷

```
Status Surface::visit(Suspend &s) {
  // to take care of immediate and counters !
  Enter *enimleaf;
  Test *tst;
  GRCNode *start;
  StartCounter *scnt;

   start = synthesize(s.body);

   Delay *d = dynamic_cast<Delay*>(s.predicate);
   if(d && d->is_immediate){
     enimleaf=new Enter();
     enimleaf->st = (STNode*) environment.grc2st[(GRCNode*)&s]->successors.back();
     *enimleaf >> context(1);
     tst = new Test(d->predicate);
     tst->st = NULL;
     *tst >> start >> enimleaf;
     start = tst;
        std::cerr<<"immediate suspend\n";
   }
   if(d && !d->is_immediate){ // a counter
        scnt = new StartCounter(d->predicate, d->count, d->counter);
          scnt->st =  (STNode*) environment.grc2st[(GRCNode*)&s]->successors[0];
        push_onto(start, new Action(scnt));
   }

  context(0) = start;

  return Status();
}
```

13 ⟨*depth method definitions* 5e⟩+≡ (26d) ◁11b 17▷

```
Status Depth::visit(Suspend &s) {
  Switch *swimm;
  Enter *enimleaf;
  Expression *pred;
  Test *t;
  GRCNode *start;

  assert(s.predicate);
  assert(s.body);

  swimm = new Switch();
    swimm->st = environment.grc2st[(GRCNode*)&s];

  start = synthesize(s.body);

  Delay *d = dynamic_cast<Delay*>(s.predicate);
  if(d) if(d->is_immediate) pred = d->predicate;
           else pred =  new CheckCounter(environment.boolean_type, d->counter);
     else pred = s.predicate;

  // the depth test: body is already started

   t = new Test(pred);
      t->st = (STNode*) environment.grc2st[(GRCNode*)&s]->successors[0];
   *t>>start>>context(1);
   *swimm >> t;

  // now the surface one
  if(d && d->is_immediate){
    enimleaf = new Enter();
    enimleaf->st = (STNode*) environment.grc2st[(GRCNode*)&s]->successors.back();
    *enimleaf >> context(1);
    t = new Test(pred);
      t->st = NULL;
    start = environment.surface.recurse(s.body);
    *t >> start >>enimleaf;
    *swimm >> t;
  }

  context(0) = swimm;
  return Status();
}
```

14a ⟨*st method definitions* 6a⟩+≡ (26d) ◁10e 19a▷

```
Status SelTree::visit(Suspend &s)
{
  STref *sp;
  STexcl *ex;
  Delay *d;

  ex=new STexcl(); environment.grc2st[(GRCNode*)&s] = ex;
  sp=new STref(); sp->setsuspend();

  *ex >> sp;
  d = dynamic_cast<Delay*>(s.predicate);
  if(d && d->is_immediate) *ex >> new STleaf();

  assert(s.body);
  *sp >> synthesize(s.body);
  context(0) = ex;
  return Status();
}
```

1.12 Abort

14b ⟨*surface methods* 5b⟩+≡ (4b) ◁11c 19b▷

```
Status visit(Abort &);
```

14c ⟨*depth methods* 5d⟩+≡ (4c) ◁11d 19c▷

```
Status visit(Abort &);
```

14d ⟨*st methods* 5f⟩+≡ (5a) ◁11e 19d▷

```
Status visit(Abort &);
```

15 ⟨*surface method definitions* 5c⟩+≡ (26d) ◁12 21▷

```
Status Surface::visit(Abort &s) {
 // Leave *lv;
  Enter *en;
  RecT1 *rt1;
  Nop *nop, *flowin;
  GRCNode *start;
  StartCounter *scnt;

//  no leaves any more !
//  lv=new Leave(); lv->st=environment.grc2st[(GRCNode*)&s];
//  push_onto(context(0), lv);

  if (s.is_weak) {
    assert(0);
  } else {

    context.push();

    rt1 = new RecT1();
    flowin = new Nop(); flowin->setflowin();
    *flowin >> rt1;
    en = new Enter();
    en ->st = (STNode*) environment.grc2st[(GRCNode*)&s]->successors.back();

    *rt1 >> en; *en >> context(1);

     for ( int i = 0 ; i < context.size ; i++ ){
        if(i==1) continue;
        nop = new Nop(); nop->code = i;
        *nop >> rt1;
        push_onto(context(i), nop);
     }

    assert(s.body);
    start = recurse(s.body);

    context.pop();
    push_onto(start, flowin);

    for ( vector<PredicatedStatement*>::reverse_iterator i = s.cases.rbegin() ;
          i != s.cases.rend() ; i++ ) {
      assert(*i);
      Delay *d = dynamic_cast<Delay*>((*i)->predicate);
      if (d) {
        if (d->is_immediate) {

          // An immediate predicate: add a test an a handler

          assert(d->predicate);
```

```
          Test *tst = new Test(d->predicate);
             tst->st = (STNode*)environment.grc2st[(GRCNode*)&s]->successors.back();
          *tst >> start;
          if ((*i)->body)
            *tst >> recurse((*i)->body);
          else
            *tst >> context(0);
          start = tst;

        } else {

          // A counted predicate: add code that initializes the counter

          if (d->count) {
            assert(d->counter);
            scnt = new StartCounter(d->predicate, d->count, d->counter);
                scnt -> st = (STNode*)environment.grc2st[(GRCNode*)&s]->successors.back();
            push_onto(start, new Action(scnt));
          }
        }
      }
    }

      context(0) = start;
//    en=new Enter(); en->st=environment.grc2st[(GRCNode*)&s];
//    push_onto(context(0), en);
  }

  return Status();
}
```

17 ⟨*depth method definitions* 5e⟩+≡ (26d) ◁13 23▷

```
Status Depth::visit(Abort &s) {
//  Leave *lv;

  Enter *en;
  RecT1 *rt1;
  Nop *nop, *flowin;
  GRCNode *resume, *hflow;
  Expression *pred;

//  lv=new Leave(); lv->st=environment.grc2st[(GRCNode*)&s];
//  push_onto(context(0), lv);

  if (s.is_weak) {
    assert(0);
  } else {

    context.push();

    rt1 = new RecT1();
    flowin = new Nop(); flowin->setflowin();
    *flowin >> rt1;
    en = new Enter();
    en ->st = (STNode*) environment.grc2st[(GRCNode*)&s]->successors.back();

    *rt1 >> en; *en >> context(1);

     for ( int i = 0 ; i < context.size ; i++ ){
        if(i==1)continue;
        nop = new Nop(); nop -> code = i;
        *nop >> rt1;
        push_onto(context(i), nop);
     }

    Switch *switch_s = new Switch();
    switch_s->st= environment.grc2st[(GRCNode*)&s];

    assert(s.body);
    resume = recurse(s.body);
//    hflow = context(1);
//    push_onto(hflow, flowin);
     hflow = flowin;

    context.pop();

    for ( vector<PredicatedStatement*>::reverse_iterator i = s.cases.rbegin() ;
          i != s.cases.rend() ; i++ ) {
        assert(*i);
        assert((*i)->predicate);
```

```
        Delay *d = dynamic_cast<Delay*>((*i)->predicate);
        if(d) if(d->is_immediate) pred = d->predicate;
                else pred = new CheckCounter(environment.boolean_type, d->counter);
          else pred = (*i)->predicate;

        // test for body activation

        Test *tst = new Test(pred);
          tst->st = (STNode*)environment.grc2st[(GRCNode*)&s];

        *tst >> resume >> context(1);
        resume = tst;

        // test for handler activation
        GRCNode *handler =
          ((*i)->body) ? environment.surface.recurse((*i)->body) : context(0);
        tst = new Test(pred);
          tst->st = (STNode*)environment.grc2st[(GRCNode*)&s];
        *tst >> hflow >> handler;
        hflow = tst;

        // Attach the depth of this handler to the main switch

        if ((*i)->body) {
                *switch_s >> recurse((*i)->body);
        }
    }

    Nop *shnop = new Nop(); shnop->setshortcut();
    *shnop >> resume >> hflow;
    *switch_s >> shnop;

    context(0) = switch_s;

  }
  return Status();
}
```

19a ⟨*st method definitions* 6a⟩+≡ (26d) ◁14a 20▷

```
Status SelTree::visit(Abort &s){
  STexcl *absw;
  STref *pre;

  absw=new STexcl();
  pre=new STref(); pre->setabort();

  environment.grc2st[(GRCNode*)&s] = absw; // the switch will point to the exclusive
  assert(s.body);

  std::cerr<<"Abort has "<<s.cases.size()<<" cases\n";
  for ( vector<PredicatedStatement*>::reverse_iterator i = s.cases.rbegin() ;
          i != s.cases.rend() ; i++ ) {
      assert(*i);
      assert((*i)->predicate);

      if((*i)->body) *absw >> synthesize((*i)->body);
        else std::cerr<<"what handler is this?\n";
  }

  *pre >> synthesize(s.body);
  *absw >> pre;
  context(0) = absw;
  return Status();
}
```

1.13 Parallel

19b ⟨*surface methods* 5b⟩+≡ (4b) ◁14b 24a▷

```
Status visit(ParallelStatementList &);
```

19c ⟨*depth methods* 5d⟩+≡ (4c) ◁14c 24b▷

```
Status visit(ParallelStatementList &);
```

19d ⟨*st methods* 5f⟩+≡ (5a) ◁14d 24c▷

```
Status visit(ParallelStatementList &);
```

20 ⟨*st method definitions* 6a⟩+≡ (26d) ◁19a 25b▷

```
Status SelTree::visit(ParallelStatementList &s)
{
  STpar *par;
  STexcl *ex;
  STleaf *term;

  par=new STpar();
  environment.grc2st[(GRCNode*)&s] = par;

  for ( vector<Statement*>::iterator i = s.threads.begin() ;
        i != s.threads.end() ; i++ ) {
    assert(*i);
    ex=new STexcl(); *par >> ex;
    term=new STleaf(); term->setfinal(); *ex >> term;
    *ex>>synthesize(*i);
  }

  context(0)=par;
  std::cerr<<"Parallel ok"<<std::endl;
  return Status();
}
```

21 ⟨*surface method definitions* 5c⟩+≡ (26d) ◁15 24d▷

```
Status Surface::visit(ParallelStatementList &s) {
  Fork *fork = new Fork();
  Sync *sync = new Sync();
  Leave *lv;
  Enter *en;
  Terminate *t0;
  int nthr, have_t0;

  push_onto(context(0), lv=new Leave());
  lv->st=environment.grc2st[(GRCNode*)&s];
  sync->st=environment.grc2st[(GRCNode*)&s];

  GRCNode **outer = context.continuations.top();
  assert(outer);
  context.push();

  // Create a new terminate for every possible exit level
  // and link each from the sync node

  for ( int i = 1 ; i < context.size ; i++ )
    context(i) = new Terminate(i);

  // Synthesize each thread's surface
  have_t0 = 0;
  for ( vector<Statement*>::iterator i = s.threads.begin() ;
        i != s.threads.end() ; i++ ) {
    assert(*i);
    nthr=i-s.threads.begin();

    // this is the self looping enter
    en=new Enter(); en->st=(STNode*)environment.grc2st[(GRCNode*)&s]->successors[nthr]->successors[0];
    t0 = new Terminate(0);
    *en >> t0;

    context(0)= en;
    *fork >> recurse(*i); // it links thread to terminates, but each thread should have its own "enter"
    if(!context(0)->predecessors.empty()) {
      // add t0 to sync here, so a sync can have a t0 for each thread
      *t0 >> sync;
      have_t0 = 1;
    }

  }


  // Connect each Terminate node with predecessors (i.e., that was
  // used by the threads) to the Sync and delete the rest.

  if(have_t0) *sync >> outer[0];
```

```
  for ( int i = 1 ; i < context.size ; i++ )
    if (context(i)) {
      if(!context(i)->predecessors.empty()) {
        *context(i) >> sync;
        *sync >> outer[i];
      } else {
        *sync >> 0;
        delete context(i);
      }
    }

  context.pop();
  context(0) = fork;
  push_onto(context(0), en=new Enter());
  en->st=environment.grc2st[(GRCNode*)&s];
  return Status();
}
```

23 ⟨*depth method definitions* 5e⟩+≡ (26d) ◁17 25a▷

```
Status Depth::visit(ParallelStatementList &s) {
  Fork *fork = new Fork();
  Sync *sync = new Sync();
  Leave *lv;
  Enter *en;
  int nthr, have_t0;
  Terminate *t0;

  push_onto(context(0), lv = new Leave());
  lv->st=environment.grc2st[(GRCNode*)&s];
  sync->st=environment.grc2st[(GRCNode*)&s];
  sync->setdepth();

  GRCNode **outer = context.continuations.top();
  assert(outer);
  context.push();

  // Create a new terminate for every possible exit level
  // and link each from the sync node

  for ( int i = 1 ; i < context.size ; i++ )
    context(i) = new Terminate(i);

  // Synthesize each thread's surface
  have_t0=0;
  for ( vector<Statement*>::iterator i = s.threads.begin() ;
        i != s.threads.end() ; i++) {
    assert(*i);
    nthr=i-s.threads.begin();
    Switch *sw = new Switch();
    sw->st=(STNode*)(environment.grc2st[(GRCNode*)&s])->successors[nthr];
    t0=new Terminate(0);
    *fork >> sw;
    // this is the self looping enter
    en=new Enter();
    en->st=(STNode*)(environment.grc2st[(GRCNode*)&s]->successors[nthr]->successors[0]);    *sw >> en;
    *en >> t0;
    context(0)=en;
    *sw >> recurse(*i);
    if(!context(0)->predecessors.empty()) {
      // add t0 to sync here, so a sync can have a t0 for each thread
      *t0 >> sync;
      have_t0 = 1;
    }
  }

  // Connect each Terminate node with predecessors (i.e., that was
  // used by the threads) to the Sync and delete the rest.
```

```
  if(have_t0) *sync >> outer[0];
  for ( int i = 1 ; i < context.size ; i++ )
    if (context(i)) {
      if(!context(i)->predecessors.empty()) {
        *context(i) >> sync;
        *sync >> outer[i];
      } else {
        *sync >> 0;
        delete context(i);
      }
    }

  context.pop();
  context(0) = fork;
  return Status();
}
```

1.14 Trap

24a ⟨*surface methods* 5b⟩+≡ (4b) ◁19b 25c▷

```
Status visit(Trap &);
```

24b ⟨*depth methods* 5d⟩+≡ (4c) ◁19c 25d▷

```
Status visit(Trap &);
```

24c ⟨*st methods* 5f⟩+≡ (5a) ◁19d 25e▷

```
Status visit(Trap &);
```

FIXME: Handle more complicated handlers

24d ⟨*surface method definitions* 5c⟩+≡ (26d) ◁21 25f▷

```
Status Surface::visit(Trap &s) {
  Leave *lv;
  Enter *en;

  lv=new Leave(); lv->st=environment.grc2st[(GRCNode*)&s];
  push_onto(context(0), lv);
  for (SymbolTable::iterator i = s.symbols->begin() ;
       i != s.symbols->end() ; i++) {
    TrapSymbol *ts = dynamic_cast<TrapSymbol*>(*i);
    assert(ts);
    context(ts->code) = context(0);
  }

  assert(s.body);
  context(0) = synthesize(s.body);
  en=new Enter(); en->st=environment.grc2st[(GRCNode*)&s];
  push_onto(context(0), en);
  return Status();
}
```

25a ⟨*depth method definitions* 5e⟩+≡ (26d) ◁23 26a▷

```
Status Depth::visit(Trap &s) {
  Leave *lv;
  lv=new Leave(); lv->st= environment.grc2st[(GRCNode*)&s];
  push_onto(context(0), lv);
  for (SymbolTable::iterator i = s.symbols->begin() ;
       i != s.symbols->end() ; i++) {
    TrapSymbol *ts = dynamic_cast<TrapSymbol*>(*i);
    assert(ts);
    context(ts->code) = context(0);
  }

  assert(s.body);
  context(0) = synthesize(s.body);
  return Status();
}
```

25b ⟨*st method definitions* 6a⟩+≡ (26d) ◁20 26b▷

```
Status SelTree::visit(Trap &s){
  STref *tr;
  tr=new STref();
  environment.grc2st[(GRCNode*)&s] = tr;
  *tr >> synthesize(s.body);
  context(0) = tr;
  return Status();
}
```

1.15 Signal

FIXME:

25c ⟨*surface methods* 5b⟩+≡ (4b) ◁24a

```
Status visit(Signal &);
```

25d ⟨*depth methods* 5d⟩+≡ (4c) ◁24b

```
Status visit(Signal &);
```

25e ⟨*st methods* 5f⟩+≡ (5a) ◁24c

```
Status visit(Signal &);
```

25f ⟨*surface method definitions* 5c⟩+≡ (26d) ◁24d

```
Status Surface::visit(Signal &s) {
  //  push_onto(context(0), new UndefineSignal(s));
  context(0) = synthesize(s.body);
  push_onto(context(0), new DefineSignal(s));
  return Status();
}
```

26a ⟨*depth method definitions* 5e⟩+≡ (26d) ◁25a

```
Status Depth::visit(Signal &s) {
  context(0) = synthesize(s.body);
  return Status();
}
```

26b ⟨*st method definitions* 6a⟩+≡ (26d) ◁25b

```
Status SelTree::visit(Signal &s) {
  context(0) = synthesize(s.body);
  push_onto(context(0), new STref());
  return Status();
}
```

2 Astgrc.hpp and .cpp

26c ⟨*ASTGRC.hpp* 26c⟩≡

```
#ifndef _ASTGRC_HPP
#  define _ASTGRC_HPP

#  include "AST.hpp"
#  include <assert.h>
#  include <stack>

namespace ASTGRC {
  using namespace IR;
  using namespace AST;

  class GrcSynth;

  ⟨context class 1⟩
  ⟨grc walker class 4a⟩
  ⟨surface class 4b⟩
  ⟨depth class 4c⟩
  ⟨st class 5a⟩
  ⟨GrcSynth class 2a⟩
}
#endif
```

26d ⟨*ASTGRC.cpp* 26d⟩≡

```
#include "ASTGRC.hpp"

namespace ASTGRC {
  ⟨surface method definitions 5c⟩
  ⟨depth method definitions 5e⟩
  ⟨st method definitions 6a⟩
}
```