CEC High-level Statement Dismantlers

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

Contents

1 Helpers

2a ⟨*fresh label declaration* 2a⟩≡ (31a)

```
Label *fresh_label();
```

2b ⟨*fresh label definition* 2b⟩≡ (31b)

```
Label *fresh_label() {
  static int nextLabel = 0;
  std::ostringstream os;
  os << "L" << nextLabel++;
  return new Label(os.str());
}
```

2c ⟨*new unique var definition* 2c⟩≡ (31b)

```
VariableSymbol *Dismantler2::new_unique_var(string basename, TypeSymbol *ts) {
  assert(ts);
  assert(module);
  assert(module->variables);
  string name = basename;
  char ch = 'a';
  while (module->variables->contains(name))
    name = basename + '_' + ch++;
  VariableSymbol *vs = new VariableSymbol(name, ts, 0);
  module->variables->enter(vs);
  return vs;
}
```

2d ⟨*add new thread definition* 2d⟩≡ (31b)

```
void Dismantler2::add_new_thread(Parallel *par, Statement *body) {
  assert(integer_type);
  Label *catch0 = fresh_label();
  Label *cont = fresh_label();
  VariableSymbol *level_var = new_unique_var("level", integer_type);
  par->newThread(catch0, cont, body, level_var);
}
```

2 Rewriting Class

By itself, this class simply does a depth-first walk of the AST; it is meant as a base class for rewriting classes that actually do something.

3a ⟨*rewriter class* 3a⟩≡ (31a)
```
class Rewriter : public Visitor {
protected:
  Module *module;
public:
  template <class T> T* transform(T* n) {
    T* result = n ? dynamic_cast<T*>(n->welcome(*this).n) : 0;
    assert(result || !n);
    return result;
  }

  template <class T> void rewrite(T* &n) { n = transform(n); }

  StatementList& sl() { return *(new StatementList()); }

  Rewriter() : module(0) {}

  ⟨rewriter methods 3b⟩
};
```

2.1 Composite Statements

These call rewrite on each of their children (e.g., bodies).

3b ⟨*rewriter methods* 3b⟩≡ (3a) 3c ▷
```
Status visit(Modules &m) {
  for (vector<Module*>::iterator i = m.modules.begin() ;
       i != m.modules.end() ; i++ ) {
    rewrite(*i);
    assert(*i);
  }
  return &m;
}
```

3c ⟨*rewriter methods* 3b⟩+≡ (3a) ◁ 3b 4a ▷
```
Status visit(Module &m) {
  module = &m;
  rewrite(m.body);
  assert(m.body);
  return &m;
}
```

4a ⟨*rewriter methods* 3b⟩+≡ (3a) ◁3c 4b▷

```
Status visit(StatementList &l) {
  for (vector<Statement*>::iterator i = l.statements.begin() ;
       i != l.statements.end() ; i++ ) {
    rewrite(*i);
    assert(*i);
  }
  return &l;
}
```

4b ⟨*rewriter methods* 3b⟩+≡ (3a) ◁4a 4c▷

```
Status visit(ParallelStatementList &l) {
  for (vector<Statement*>::iterator i = l.threads.begin() ;
       i != l.threads.end() ; i++ ) {
    rewrite(*i);
    assert(*i);
  }
  return &l;
}
```

4c ⟨*rewriter methods* 3b⟩+≡ (3a) ◁4b 5a▷

```
Status visit(Loop &s) {
  rewrite(s.body);
  return &s;
}

Status visit(Repeat &s) {
  rewrite(s.count);
  rewrite(s.body);
  return &s;
}

Status visit(Suspend &s) {
  rewrite(s.predicate);
  rewrite(s.body);
  return &s;
}

Status visit(Abort &s) {
  rewrite(s.body);
  for (vector<PredicatedStatement*>::iterator i = s.cases.begin() ;
       i != s.cases.end() ; i++) {
    assert(*i);
    rewrite(*i);
  }
  return &s;
}
```

5a ⟨*rewriter methods* 3b⟩+≡ (3a) ◁4c 5b▷

```
Status visit(PredicatedStatement &s) {
  rewrite(s.predicate);
  rewrite(s.body);
  return &s;
}
```

5b ⟨*rewriter methods* 3b⟩+≡ (3a) ◁5a 5c▷

```
Status visit(Trap &s) {
  rewrite(s.body);
  for (vector<PredicatedStatement*>::iterator i = s.handlers.begin() ;
       i != s.handlers.end() ; i++) {
    assert(*i);
    rewrite((*i)->body);
  }
  return &s;
}
```

5c ⟨*rewriter methods* 3b⟩+≡ (3a) ◁5b 5d▷

```
Status visit(IfThenElse& n) {
  rewrite(n.predicate);
  rewrite(n.then_part);
  rewrite(n.else_part);
  return &n;
}
```

5d ⟨*rewriter methods* 3b⟩+≡ (3a) ◁5c 5e▷

```
Status visit(Handler& s) {
  for ( vector<Catch*>::iterator i = s.catches.begin() ;
        i != s.catches.end() ; i++ ) {
    assert(*i);
    rewrite((*i)->body);
  }
  return &s;
}
```

5e ⟨*rewriter methods* 3b⟩+≡ (3a) ◁5d 5f▷

```
Status visit(Try& s) {
  rewrite(s.body);
  Handler &h = s;
  return visit(h);
}
```

5f ⟨*rewriter methods* 3b⟩+≡ (3a) ◁5e 6a▷

```
Status visit(Resume& s) {
  rewrite(s.body);
  Handler &h = s;
  return visit(h);
}
```

6a ⟨*rewriter methods* 3b⟩+≡ (3a) ◁5f 6b▷
```
Status visit(Thread& s) {
  Resume &r = s;
  return visit(r);
}
```

6b ⟨*rewriter methods* 3b⟩+≡ (3a) ◁6a 6c▷
```
Status visit(Signal& s) {
  rewrite(s.body);
  return &s;
}
```

6c ⟨*rewriter methods* 3b⟩+≡ (3a) ◁6b 6d▷
```
Status visit(Var& s) {
  rewrite(s.body);
  return &s;
}
```

6d ⟨*rewriter methods* 3b⟩+≡ (3a) ◁6c 6e▷
```
Status visit(Parallel& s) {
  for ( vector<Thread*>::iterator i = s.threads.begin() ;
        i != s.threads.end() ; i++ ) {
    assert(*i);
    rewrite(*i);
  }
  Handler &h = s;
  return visit(h);
}
```

6e ⟨*rewriter methods* 3b⟩+≡ (3a) ◁6d 6f▷
```
Status visit(ProcedureCall& s) {
  for ( vector<Expression*>::iterator i = s.value_args.begin() ;
        i != s.value_args.end() ; i++ ) {
    assert(*i);
    rewrite(*i);
  }
  return &s;
}
```

6f ⟨*rewriter methods* 3b⟩+≡ (3a) ◁6e 6g▷
```
Status visit(Emit& s) {
  rewrite(s.value);
  return &s;
}
```

6g ⟨*rewriter methods* 3b⟩+≡ (3a) ◁6f 7a▷
```
Status visit(Assignment& s) {
  rewrite(s.value);
  return &s;
}
```

7a ⟨*rewriter methods* 3b⟩+≡ (3a) ◁6g 7b▷
```
Status visit(Catch &s) {
  rewrite(s.body);
  return &s;
}
```

7b ⟨*rewriter methods* 3b⟩+≡ (3a) ◁7a 7c▷
```
Status visit(Present& s) {
  for ( vector<PredicatedStatement*>::iterator i = s.cases.begin() ;
        i != s.cases.end() ; i++ ) {
    assert(*i);
    rewrite(*i);
  }
  if (s.default_stmt) rewrite(s.default_stmt);
  return &s;
}
```

7c ⟨*rewriter methods* 3b⟩+≡ (3a) ◁7b 7d▷
```
Status visit(If& s) {
  for ( vector<PredicatedStatement*>::iterator i = s.cases.begin() ;
        i != s.cases.end() ; i++ ) {
    assert(*i);
    rewrite(*i);
  }
  if (s.default_stmt) rewrite(s.default_stmt);
  return &s;
}
```

2.2 Leaf Statements

These stop the recursion and return themselves;

7d ⟨*rewriter methods* 3b⟩+≡ (3a) ◁7c 8a▷
```
Status visit(Nothing& n) { return &n; }
Status visit(Pause& n) { return &n; }
Status visit(Halt& n) { return &n; }
Status visit(Sustain& n) { return &n; }
Status visit(Await& n) { return &n; }
Status visit(LoopEach& n) { return &n; }
Status visit(Every& n) { return &n; }
Status visit(DoWatching& n) { return &n; }
Status visit(DoUpto& n) { return &n; }
Status visit(TaskCall& n) { return &n; }
Status visit(Exec& n) { return &n; }
Status visit(Exit& n) { return &n; }
Status visit(Run& n) { return &n; }

Status visit(Label& n) { return &n; }
Status visit(Goto& n) { return &n; }
Status visit(Break& n) { return &n; }
Status visit(Continue& n) { return &n; }
```

2.3 Expressions

8a ⟨*rewriter methods* 3b⟩+≡ (3a) ◁7d 8b▷

```
Status visit(LoadVariableExpression &e) { return &e; }
Status visit(LoadSignalExpression &e) { return &e; }
Status visit(LoadSignalValueExpression &e) { return &e; }
Status visit(LoadTrapExpression &e) { return &e; }
Status visit(LoadTrapValueExpression &e) { return &e; }
Status visit(Literal &e) { return &e; }
```

8b ⟨*rewriter methods* 3b⟩+≡ (3a) ◁8a 8c▷

```
Status visit(UnaryOp &e) {
  rewrite(e.source);
  return &e;
}
```

8c ⟨*rewriter methods* 3b⟩+≡ (3a) ◁8b 8d▷

```
Status visit(BinaryOp &e) {
  rewrite(e.source1);
  rewrite(e.source2);
  return &e;
}
```

8d ⟨*rewriter methods* 3b⟩+≡ (3a) ◁8c 8e▷

```
Status visit(FunctionCall &e) {
  for ( vector<Expression*>::iterator i = e.arguments.begin() ;
        i != e.arguments.end() ; i++ ) {
    assert(*i);
    rewrite(*i);
  }
  return &e;
}
```

8e ⟨*rewriter methods* 3b⟩+≡ (3a) ◁8d

```
Status visit(Delay &e) {
  rewrite(e.predicate);
  rewrite(e.count);
  return &e;
}
```

3 First Pass

This uses the `Rewriter` class to perform a preorder traversal of the tree of statements in each module to rewrite each node as it goes. After a method has dismantled its object, it calls `rewrite` on itself to insure things are dismantled as far as possible.

9a ⟨*first pass class* 9a⟩≡ (31a)

```
class Dismantler1 : public Rewriter {
public:
  ⟨first pass methods 9b⟩
};
```

3.1 Case Statements: Present and If

Present and If statements are dismantled into a cascade of if-then-else statements:

```
present                        if (p1) s1
  case p1 do s1                else if (p2) s2
  case p2 do s2                     else s3
  else s3
end
```

9b ⟨*first pass methods* 9b⟩≡ (9a) 10a ▷

```
IfThenElse *dismantle_case(CaseStatement &c) {
  assert(c.cases.size() > 0);
  IfThenElse *result = 0;
  IfThenElse *lastif = 0;

  for (vector<PredicatedStatement*>::iterator i = c.cases.begin() ;
       i != c.cases.end() ; i++ ) {
    assert(*i);
    assert((*i)->predicate);
    IfThenElse *thisif = new IfThenElse((*i)->predicate);
    thisif->then_part = transform((*i)->body);
    if (result)
      lastif->else_part = thisif;
    else
      result = thisif;
    lastif = thisif;
  }
  assert(lastif);
  lastif->else_part = c.default_stmt;
  assert(result);
  return transform(result);
}

virtual Status visit(Present &s) { return dismantle_case(s); }
virtual Status visit(If &s) { return dismantle_case(s); }
```

3.2 Await

Await becomes an *abort* running a halt statement.

```
await
  case immediate p1 do s1
  case p2 do s2
  case p3 do s3
end
```

```
abort
  loop pause end
when
  case immediate p1 do s1
  case p2 do s2
  case p3 do s3
end
```

10a ⟨*first pass methods* 9b⟩+≡ (9a) ◁9b 10b▷

```
Status visit(Await &a) {
  Pause *p = new Pause();
  Loop *l = new Loop(p);
  Abort *ab = new Abort(l, false);
  // Copy the predicates
  for ( vector<PredicatedStatement*>::const_iterator i = a.cases.begin();
        i != a.cases.end() ; i++ )
    ab->cases.push_back(*i);
  return transform(ab);
}
```

3.3 Do Watching and Do Upto

```
do
  b
watching p timeout s
```

```
abort
  b
when p do s
```

10b ⟨*first pass methods* 9b⟩+≡ (9a) ◁10a 10c▷

```
Status visit(DoWatching &s) {
  return transform(new Abort(s.body, s.predicate, s.timeout));
}
```

```
do
  b
upto p
```

```
abort
  b ;
  halt
when p
```

10c ⟨*first pass methods* 9b⟩+≡ (9a) ◁10b 11▷

```
Status visit(DoUpto &s) {
  return transform(new Abort(&(sl() << s.body << new Halt()), s.predicate, 0));
}
```

3.4 Loop Each

loop
    b
each p
	loop
	  abort
	    b ;
	    halt
	  when p
	end

11 ⟨*first pass methods* 9b⟩+≡ (9a) ◁10c 12a▷

```
  Status visit(LoopEach &s) {
    return transform(new Loop(new Abort(&(sl() << s.body << new Halt()),
                                        s.predicate, 0)));
  }
```

3.5 Every

```
every          goto Every
  p            loop
do b             abort
                   b;
                   Every:;
                   halt
                 when p
               end

               abort
                 halt
               when p; -- keeps immediate attribute
               loop
                abort
                 b;
                 halt
                when p -- remove the immediate attribute
               end
```

12a ⟨*first pass methods* 9b⟩+≡ (9a) ◁11 12b▷

```
Status visit(Every &s) {
/*  Label *l = fresh_label();

  return transform(&(sl() << new Goto(l) <<
                   new Loop(new Abort(&(sl() << s.body << l << new Halt()),
                                      s.predicate, 0))));
*/
  Expression *pred_secondabort;
  Delay *d = dynamic_cast<Delay*>(s.predicate);

  pred_secondabort = (d && d->is_immediate) ? d->predicate : s.predicate;

  return transform(&(sl() << new Abort(new Halt(), s.predicate, 0) <<
        new Loop(new Abort(&(sl()<< s.body << new Halt()), pred_secondabort, 0))
        ));
}
```

3.6 Halt

```
halt                                  loop
                                        pause
                                      end
```

12b ⟨*first pass methods* 9b⟩+≡ (9a) ◁12a 13a▷

```
Status visit(Halt &s) {
  return transform(new Loop(new Pause()));
}
```

3.7 Sustain

sustain s	**loop**
	  emit s ;
	  pause
	end

13a ⟨*first pass methods* 9b⟩+≡ (9a) ◁12b 13b▷

```
Status visit(Sustain &s) {
  return transform(new Loop(&(sl() <<
                             new Emit(s.signal, s.value) << new Pause())));
}
```

3.8 Nothing

A nothing statement is replaced with an empty instruction sequence.

13b ⟨*first pass methods* 9b⟩+≡ (9a) ◁13a

```
Status visit(Nothing &) {
  return transform(new StatementList());
}
```

4 Assigning Completion Codes

Before *exit* statements can be dismantled, completion codes/exit levels must be assigned throughout the program. *Weak abort* statements also use completion codes (see the dismantler for the *abort* statement), so they are also considered here.

13c ⟨*completion code class* 13c⟩≡ (31a)

```
class CompletionCodes : public Visitor {
public:
  void alsoMax(AST::ASTNode *n, int &m) {
     int max = recurse(n);
     if (max > m) m = max;
  }

  int recurse(AST::ASTNode *n) {
     if (n) return n->welcome(*this).i;
     else return 0;
  }

  ⟨completion code methods 14a⟩
};
```

4.1 Composite Statements

14a ⟨*completion code methods* 14a⟩≡ (13c) 14b ▷

```
Status visit(Modules &m) {
  int max = 1;
  for (vector<Module*>::iterator i = m.modules.begin() ;
       i != m.modules.end() ; i++ ) alsoMax(*i, max);
  return max;
}
```

14b ⟨*completion code methods* 14a⟩+≡ (13c) ◁ 14a 14c ▷

```
Status visit(Module& s) {
  s.max_code = recurse(s.body);
  return s.max_code;
}
```

14c ⟨*completion code methods* 14a⟩+≡ (13c) ◁ 14b 14d ▷

```
Status visit(Signal& s) { return recurse(s.body); }
Status visit(Var& s) { return recurse(s.body); }
Status visit(Loop &s) { return recurse(s.body); }
Status visit(Repeat &s) { return recurse(s.body); }
Status visit(Suspend &s) { return recurse(s.body); }
Status visit(PredicatedStatement &s) { return recurse(s.body); }
```

14d ⟨*completion code methods* 14a⟩+≡ (13c) ◁ 14c 14e ▷

```
Status visit(StatementList &l) {
  int max = 1;
  for (vector<Statement*>::iterator i = l.statements.begin() ;
       i != l.statements.end() ; i++ ) alsoMax(*i, max);
  return max;
}
```

14e ⟨*completion code methods* 14a⟩+≡ (13c) ◁ 14d 14f ▷

```
Status visit(ParallelStatementList &l) {
  int max = 1;
  for (vector<Statement*>::iterator i = l.threads.begin() ;
       i != l.threads.end() ; i++ ) alsoMax(*i, max);
  return max;
}
```

14f ⟨*completion code methods* 14a⟩+≡ (13c) ◁ 14e 15a ▷

```
Status visit(IfThenElse& n) {
  int max = 1;
  alsoMax(n.then_part, max);
  alsoMax(n.else_part, max);
  return max;
}
```

4.2 Leaf Statements

None of these needs a completion code, so they all return 0;

15a ⟨*completion code methods* 14a⟩+≡ (13c) ◁14f 15b▷

```
Status visit(Pause&) { return Status(0); }
Status visit(Emit&) { return Status(0); }
Status visit(Assignment&) { return Status(0); }
Status visit(ProcedureCall&) { return Status(0); }
Status visit(TaskCall&) { return Status(0); }
Status visit(Exec&) { return Status(0); }
Status visit(Exit&) { return Status(0); }
Status visit(Run&) { return Status(0); }
Status visit(Label&) { return Status(0); }
Status visit(Goto&) { return Status(0); }
```

4.3 Abort

Weak abort statements use one code for normal termination and one for each case; strong aborts do not use any more.

15b ⟨*completion code methods* 14a⟩+≡ (13c) ◁15a 16a▷

```
Status visit(Abort &s) {
  int max = 1;
  alsoMax(s.body, max);
  for (vector<PredicatedStatement*>::iterator i = s.cases.begin() ;
       i != s.cases.end() ; i++ ) alsoMax(*i, max);
  if (s.is_weak) {
    s.code = max + 1;
    assert(s.code >= 2);
    max += 1 + s.cases.size();
  }
  return max;
}
```

4.4 Trap

Trap is the only statement that consumes completion codes.

16a ⟨*completion code methods* 14a⟩+≡ (13c) ◁15b

```
Status visit(Trap &s) {
  int max = 1;
  alsoMax(s.body, max);

  // FIXME: is this the right order?  Should the predicates be
  // considered before or after the code is assigned?

  for (vector<PredicatedStatement*>::iterator i = s.handlers.begin() ;
       i != s.handlers.end() ; i++ ) alsoMax(*i, max);

  max++; // Allocate an exit level for this trap statement

  assert(s.symbols);
  for (SymbolTable::iterator i = s.symbols->begin() ; i !=
       s.symbols->end() ; i++) {
    TrapSymbol *ts = dynamic_cast<TrapSymbol*>(*i);
    assert(ts);
    ts->code = max;
  }

  return max;
}
```

5 Second Pass

Once completion codes are assigned, things can be dismanted much farther.

16b ⟨*second pass class* 16b⟩≡ (31a)

```
class Dismantler2 : public Rewriter {
  BuiltinConstantSymbol *true_constant;
  BuiltinConstantSymbol *false_constant;
  BuiltinTypeSymbol *integer_type;
  BuiltinTypeSymbol *boolean_type;
public:

  Dismantler2() : true_constant(0), false_constant(0) {}

  VariableSymbol *new_unique_var(string, TypeSymbol*);
  void add_new_thread(Parallel *, Statement *);

  ⟨second pass methods 17a⟩
};
```

17a ⟨*second pass methods* 17a⟩≡ (16b) 17b▷

```
Status visit(Module &m) {
  true_constant =
        dynamic_cast<BuiltinConstantSymbol*>(m.constants->get("true"));
  assert(true_constant);
  false_constant =
        dynamic_cast<BuiltinConstantSymbol*>(m.constants->get("false"));
  assert(false_constant);
  integer_type = dynamic_cast<BuiltinTypeSymbol*>(m.types->get("integer"));
  assert(integer_type);
  boolean_type = dynamic_cast<BuiltinTypeSymbol*>(m.types->get("boolean"));
  assert(boolean_type);

  return Rewriter::visit(m);
}
```

5.1 Pause

pause	**break** 1

17b ⟨*second pass methods* 17a⟩+≡ (16b) ◁17a 17c▷

```
Status visit(Pause &s) {
  return transform(new Break(1));
}
```

5.2 Exit

exit T(expr)	T := 1; Tvar := expr; **break** code;

17c ⟨*second pass methods* 17a⟩+≡ (16b) ◁17b 18a▷

```
Status visit(Exit &e) {
  StatementList *sl = new StatementList();
  assert(e.trap);
  assert(e.trap->code > 0);
  assert(e.trap->presence);
  assert(true_constant); // Should have been set when we visited the module
  *sl << new Assignment(e.trap->presence,
                        new LoadVariableExpression(true_constant));
  if (e.value) {
    assert(e.trap->value);
    *sl << new Assignment(e.trap->value, e.value);
  }
  *sl << new Break(e.trap->code);
  return transform(sl);
}
```

5.3 Loop

```
loop                      Restart:;
  b                       b;
end                       goto Restart
```

18a ⟨*second pass methods* 17a⟩+≡ (16b) ◁17c 18b▷

```
Status visit(Loop &s) {
  Label *l = fresh_label();
  return transform(&(sl() << l << s.body << new Goto(l)));
}
```

5.4 Emit

```
emit S(e)                 S_presence = true
                          S_value = e
```

18b ⟨*second pass methods* 17a⟩+≡ (16b) ◁18a 19▷

```
Status visit(Emit &s) {
  StatementList *sl = new StatementList();
  assert(s.signal);
  assert(s.signal->presence);
  assert(true_constant); // Should have been set up when we visited the module
  *sl << new Assignment(s.signal->presence,
                        new LoadVariableExpression(true_constant));
  if (s.value) {
    assert(s.signal->value);
    *sl << new Assignment(s.signal->value, s.value);
  }
  return transform(sl);
}
```

5.5 Statement Lists

A statement list in another statement list is merged into its parent.

19 ⟨*second pass methods* 17a⟩+≡ (16b) ◁18b 20▷

```
Status visit(StatementList &s) {
  Rewriter::visit(s);

  StatementList *result = new StatementList();

  for (vector<Statement*>::iterator i = s.statements.begin() ;
       i != s.statements.end() ; i++) {
    StatementList *sl = dynamic_cast<StatementList*>(*i);
    if (sl)
      for (vector<Statement*>::iterator j = sl->statements.begin() ;
           j != sl->statements.end() ; j++ ) *result << *j;
    else *result << *i;
  }

  return result;
}
```

5.6 Trap

```
trap T1 := e : integer in
  body
handle T1 do h1
handle T2 do h2
```

```
T1_presence = false
T1_val = e
try {
  body
} catch n {
    if (T1) { h1 }
  ||
    if (T2) { h2 }
}
```

20 ⟨*second pass methods* 17a⟩+≡ (16b) ◁19 22▷

```
Status visit(Trap &s) {
  assert(s.symbols);

  StatementList *result = new StatementList();

  int code = 0;
  int numTraps = 0;

  // Reset trap presence variables and initialize any values
  for (SymbolTable::iterator i = s.symbols->begin() ;
       i != s.symbols->end() ; i++) {
    TrapSymbol *ts = dynamic_cast<TrapSymbol*>(*i);
    assert(ts);
    code = ts->code;
    ++numTraps;

    // Reset the trap presence variable

    assert(false_constant); // Should have been set when module was visited
    *result << new Assignment(ts->presence,
                              new LoadVariableExpression(false_constant));

    // Initialize the trap value variable if specified
    if (ts->initializer) {
      assert(ts->value); // Traps with initializers should have values
      *result << new Assignment(ts->value, ts->initializer);
    }
  }

  // Need to check predicates if there's more than one trap
  bool need_tests = numTraps > 1;

  // Handlers run in parallel if there's more than one
  bool in_parallel = s.handlers.size() > 1;

  Statement *catchBody = 0;
```

```
  ParallelStatementList *psl;
  if (in_parallel) {
    catchBody = psl = new ParallelStatementList();
  }

  if (s.handlers.size() == 0)
    catchBody = new StatementList();
  else
    for (vector<PredicatedStatement*>::const_iterator i = s.handlers.begin() ;
         i != s.handlers.end() ; i++) {
      PredicatedStatement *handler = *i;
      assert(handler);
      Statement *body = handler->body;
      if (need_tests)
        body = new IfThenElse(handler->predicate, body, 0);
      if (in_parallel) psl->threads.push_back(body);
      else catchBody = body;
    }
  assert(catchBody);

  Try *tryst = new Try(fresh_label(), s.body);
  tryst->newCatch(catchBody, code, fresh_label());
  *result << tryst;

  return transform(result);
}
```

5.7 Abort

One of the most complicated since it must handle both strong and weak abort statements, immediate and counted delays.

```
abort
  body
when
  case immediate p1 do s1
  case p2 do s2
  case count p3 do s3
end
```

```
if (p1) goto S1
c3 = e3
resume {
  body
} catch 1 {
  break 1
  if (p1) { S1: s1 }
  else if (p2) { s2 }
  else if (p3) {
    cs := cs - 1;
    if (cs <= 0) { s3 }
    else goto CountCont0
  } else {
    CountCont0:
    continue
  }
}
```

FIXME: re-examine weak abort dismantling code

22 ⟨*second pass methods* 17a⟩+≡ (16b) ◁20 26a▷

```
Status visit(Abort &s) {
  Parallel *parallel;
  assert(boolean_type); // Module visitor should have found it
  assert(integer_type); // Module visitor should have found it

  if (!s.is_weak) parallel = 0;
  else {
    Label *c0 = fresh_label(); // Separated for determinism
    Label *cont = fresh_label();
    parallel = new Parallel(c0, cont);

    StatementList *catch1 = new StatementList();
    *catch1 << new Break(1) << new Continue();
    parallel->newCatch(catch1, 1, fresh_label());
  }

  // For immediate tests and counter initializations
  StatementList *preamble = new StatementList();

  int code = s.code;
  assert(!s.is_weak || code >= 2); // Weak abort should have a code

  IfThenElse *firstif = 0;
  IfThenElse *previousif = 0;
  Label *elselabel = 0;
```

```
Label *nextelselabel = 0;

for (vector<PredicatedStatement*>::const_iterator i = s.cases.begin() ;
     i != s.cases.end() ; i++) {
  PredicatedStatement *ps = *i;
  assert(ps);
  Expression *e = ps->predicate;
  assert(e);
  Statement *handler = ps->body;

  if (s.is_weak) {
    if (!handler) handler = new StatementList();
    parallel->newCatch(handler, code, fresh_label());
    handler = new Break(code);
  }

  Delay *d = dynamic_cast<Delay*>(e);
  if (d) {
    if (d->is_immediate) {
      e = d->predicate;
      assert(e);
      StatementList *newhandler = new StatementList();
      Label *handler_label = fresh_label();
      *newhandler << handler_label;
      *preamble << new IfThenElse(e, new Goto(handler_label), 0);
      if (handler) *newhandler << handler;
      handler = newhandler;
      // Note: expression e is shared
    }

    if (d->count) {

      // A counted delay:
      // Create the count variable
      // Initialize it in the preamble
      // test and decrement in the expression

      assert(d->counter);

      *preamble << new Assignment(d->counter, d->count);
      StatementList *newhandler = new StatementList();
      *newhandler <<
        new Assignment(d->counter,
                       new BinaryOp(d->counter->type, "-",
                                    new LoadVariableExpression(d->counter),
                                    new Literal("1", d->counter->type)));
      // Tricky: to avoid a side-effect in an expression, we mimic C's &&
      // operator by adding a label in the main "else" branch.
      // Nextelselabel is that label, and either the next iterator of the
      // loop or the code that adds the final continue will instantiate
```

```
      // the label itself.

      nextelselabel = fresh_label();
      if (!handler) handler = new StatementList();
      Expression *checkneg =
        new BinaryOp(boolean_type, "<=",
                     new LoadVariableExpression(d->counter),
                     new Literal("0", integer_type));

      *newhandler << new IfThenElse(checkneg, handler,
                                    new Goto(nextelselabel));
      handler = newhandler;
    } else {
      nextelselabel = 0;
    }
  }

  IfThenElse *thisif = new IfThenElse(e, handler, 0);
  if (previousif) {
    if (elselabel) {
      StatementList *sl = new StatementList();
      *sl << elselabel << thisif;
      previousif->else_part = sl;
    } else {
      previousif->else_part = thisif;
    }
  } else {
    firstif = thisif;
  }
  previousif = thisif;
  elselabel = nextelselabel;

  --code;
}

Label *again_label;
if (s.is_weak) {
  again_label = fresh_label();
  *preamble << again_label << new Break(1);
}

// Add either continue or a goto to the last "else" clause

assert(firstif);
assert(previousif);
Statement *lastbody = s.is_weak ?
  (Statement*) new Goto(again_label) : (Statement*) new Continue();
if (elselabel) {
  StatementList *sl = new StatementList();
  *sl << elselabel << lastbody;
```

```
    lastbody = sl;
  }
  previousif->else_part = lastbody;

  // Add resume ( body ) catch 1 ( tests ... continue )

  assert(s.body);
  Statement *body = s.body;

  if (s.is_weak) {

    // Create the first (body) thread: add "break k" to the end
    // of the body, surround it with a resume, and add it to the parallel

    StatementList *new_body = new StatementList();
    *new_body << body << new Break(code);
    body = new_body;

    // add catch k with an empty body (k is normal termination for the body)

    parallel->newCatch(new StatementList(), code, fresh_label());

    add_new_thread(parallel, body);

    *preamble << firstif;

    add_new_thread(parallel, preamble);

    return transform(parallel);
  }

  // Strong abort

  Label *catch0 = fresh_label();
  Label *cont = fresh_label();
  Resume *resume = new Resume(catch0, cont, body);

  StatementList *catchbody = new StatementList();
  *catchbody << new Break(1) << firstif;
  resume->newCatch(catchbody, 1, fresh_label());

  *preamble << resume;

  return transform(preamble);
}
```

5.8 Expressions

A reference to a trap is replaced with a reference to its presence variable.

26a ⟨*second pass methods* 17a⟩+≡ (16b) ◁22 26b▷
```
Status visit(LoadTrapExpression &e) {
  assert(e.trap);
  assert(e.trap->presence);
  return transform(new LoadVariableExpression(e.trap->presence));
}
```

26b ⟨*second pass methods* 17a⟩+≡ (16b) ◁26a 26c▷
```
Status visit(LoadTrapValueExpression &e) {
  assert(e.trap);
  assert(e.trap->value);
  return transform(new LoadVariableExpression(e.trap->value));
}
```

26c ⟨*second pass methods* 17a⟩+≡ (16b) ◁26b 26d▷
```
Status visit(LoadSignalExpression &e) {
  assert(e.signal);
  assert(e.signal->presence);
  return transform(new LoadVariableExpression(e.signal->presence));
}
```

26d ⟨*second pass methods* 17a⟩+≡ (16b) ◁26c 27▷
```
Status visit(LoadSignalValueExpression &e) {
  assert(e.signal);
  assert(e.signal->value);
  return transform(new LoadVariableExpression(e.signal->value));
}
```

5.9 Parallel Statement Lists

b1 || b2

parallel {
  thread {
    b1
    break 0
  }
  thread {
    b2
    break 0
  }
}

27 ⟨*second pass methods* 17a⟩+≡ (16b) ◁26d

```
Status visit(ParallelStatementList &s) {
  Label *catch0 = fresh_label();
  Label *cont = fresh_label();
  Parallel *result = new Parallel(catch0, cont);

  assert(s.threads.size() > 0);

  for ( vector<Statement*>::iterator i = s.threads.begin() ;
        i != s.threads.end() ; i++ ) {
    StatementList *body = dynamic_cast<StatementList*>(*i);
    if (!body) {
      body = new StatementList();
      *body << *i;
    }
    *body << new Break(0);
    add_new_thread(result, body);
  }
  return transform(result);
}
```

6 Add Parallel Catches

At a Parallel, the exit level of each of its threads is tested and only the highest exit level is allowed to propagate. The rewriter in this section adds catch clauses to each Thread statement that catches every exit level it generates:

```
thread {

  break k

}
```

```
thread {

  break k

} catch k {
   Level_var = k
}
```

Once these exit levels are caught by the threads, each parallel must re-throw them, so this procedure also adds catch clauses that simply re-throw their exceptions to each Parallel.

```
parallel {
  thread {..}
  catch 1 {}
  catch 0 {}

  thread {..}
  catch 2 {}
}
```

```
parallel {
  thread {..}
  catch 1 {}
  catch 0 {}

  thread {..}
  catch 2 {}
} catch 2 { break 2 }
```

The body of *Catch* 0 is empty; the body of *catch* 1 includes a *continue*; all the others simply contain a *break*, i.e.,

```
catch 0 {}
catch 1 { break 1 ; continue }
catch 2 { break 2 }
catch 3 { break 3 }
```

This runs after dismantling parallel statement lists but before dismantling resume, parallel, break, etc.

28a ⟨*add parallel catches class* 28a⟩≡ (31a)

```
class ParallelCatches : public Rewriter {
  std::set<int> *levels;
public:
  ParallelCatches() : levels(0) {}
  ⟨parallel catch methods 28b⟩
};
```

The break statement simply adds its level to the set of levels for its thread.

28b ⟨*parallel catch methods* 28b⟩≡ (28a) 29 ▷

```
Status visit(Break &s) {
  if (levels) levels->insert(s.level);
  return &s;
}
```

The thread statement collects the exit levels for its body, then adds a catch clause for each. It assumes that an enclosing Parallel has created and initialized the set of levels;

29 ⟨*parallel catch methods* 28b⟩+≡ (28a) ◁28b 30▷

```
Status visit(Thread &s) {
  assert(levels);
  assert(levels->empty()); // Should have been initialized by our Parallel
  assert(s.body);
  rewrite(s.body);
  assert(s.catches.empty()); // Shouldn't have any catch clauses yet
  assert(s.exit_level); // Should have an exit level variable
  assert(s.exit_level->type); // with a type

  // Add a catch clause for each of the exit levels
  // catch k  { exit_level = k }
  for ( std::set<int>::iterator i = levels->begin() ;
        i != levels->end() ; i++) {
    std::ostringstream os;
    os << *i;
    s.newCatch(new Assignment(s.exit_level,
                              new Literal(os.str(),s.exit_level->type)),
               *i, fresh_label());
  }
  return &s;
}
```

Which completion codes may be generated by the threads within a parallel is an important question. Ultimately, the exit level is the maximum of all the threads. Over two sets L and R, this operation is defined as

$$\begin{aligned}\max(L,R) &= \{\max(l,r) \mid l \in L, r \in R\} \\ &= \{i \geq \min(L)\} \cap (L \cup R) \cap \{j \geq \min(R)\}\end{aligned}$$

The set is the union of all completion codes from the threads that are greater or equal to the highest minimum code from any thread.

FIXME: There are too many break 0 statements inserted at the end of each thread. We really want to know whether control can reach them. Exit level computation should be smarter about control flow. Berry effectively had the control-flow graph and did a careful walk based on that.

30 ⟨*parallel catch methods* 28b⟩+≡ (28a) ◁29

```
Status visit(Parallel &s) {
  std::set<int> *outer = levels;

  std::set<int> parallel_levels;

  int min = 0; // ultimately, the minimum exit level considered
  for ( unsigned int k = 0 ; k < s.threads.size() ; k++ ) {
    assert(s.threads[k]);
    std::set<int> thread_levels;
    levels = &thread_levels;
    rewrite(s.threads[k]);
    assert(!thread_levels.empty()); // should have found at least one break
    std::set<int>::const_iterator i = thread_levels.begin();
    if (*i > min) min = *i;
    for ( ; i != thread_levels.end() ; i++ ) parallel_levels.insert(*i);
  }

  // Create a catch for each member of the parallel_levels set that is
  // greater than or equal to min, the maximum of the minimum levels
  // of each of the threads

  for ( std::set<int>::const_iterator i = parallel_levels.begin() ;
        i != parallel_levels.end() ; i++ )
    if (*i >= min) {
      if (outer) outer->insert(*i);
      if (*i == 1)
        s.newCatch(&(sl() <<new Break(1) <<new Continue()), 1, fresh_label());
      else
        s.newCatch(new Break(*i), *i, fresh_label());
    }

  levels = outer;
```

```
  return &s;
};
```

7 Dismantle.hpp and .cpp

31a ⟨*Dismantle.hpp* 31a⟩≡

```
#ifndef _DISMANTLE_HPP
#  define _DISMANTLE_HPP

#  include "AST.hpp"
#  include <assert.h>
#  include <sstream>
#  include <set>

namespace Dismantle {
  using namespace IR;
  using namespace AST;

  ⟨fresh label declaration 2a⟩
  ⟨rewriter class 3a⟩
  ⟨first pass class 9a⟩
  ⟨completion code class 13c⟩
  ⟨second pass class 16b⟩
  ⟨add parallel catches class 28a⟩
}
#endif
```

31b ⟨*Dismantle.cpp* 31b⟩≡

```
#include "Dismantle.hpp"

namespace Dismantle {
  ⟨fresh label definition 2b⟩
  ⟨new unique var definition 2c⟩
  ⟨add new thread definition 2d⟩
}
```