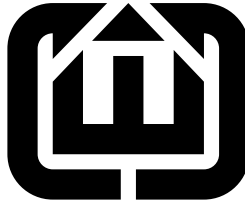


CEC High-level Statement Dismantlers



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

Contents

1	Rewriting Class	1
1.1	Composite Statements	2
1.2	Leaf Statements	5
1.3	Expressions	6
2	Statement Dismantlers	7
2.1	Case Statements: Present and If	8
2.2	Await	9
2.3	Trap	10
2.4	Weak Abort	11
2.5	Var	13
2.6	Do Watching and Do Upto	14
2.7	Loop Each	14
2.8	Halt	14
2.9	Sustain	15
2.10	Nothing	15
3	Dismantle.hpp and .cpp	15

1 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.

1 `<rewriter class 1>≡`

```

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 2a>
};

```

1.1 Composite Statements

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

- 2a *<rewriter methods 2a>*≡
- ```

Status visit(Modules &m) {
 for (vector<Module*>::iterator i = m.modules.begin() ;
 i != m.modules.end() ; i++) {
 rewrite(*i);
 assert(*i);
 }
 return &m;
}

```
- 2b *<rewriter methods 2a>*+≡
- ```

Status visit(Module &m) {
    module = &m;
    rewrite(m.body);
    assert(m.body);
    return &m;
}

```
- 2c *<rewriter methods 2a>*+≡
- ```

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

```

```

3a <rewriter methods 2a>+≡
 Status visit(ParallelStatementList &l) {
 for (vector<Statement*>::iterator i = l.threads.begin() ;
 i != l.threads.end() ; i++) {
 rewrite(*i);
 assert(*i);
 }
 return &l;
 }

3b <rewriter methods 2a>+≡
 Status visit(Loop &s) {
 rewrite(s.body);
 return &s;
 }

3c <rewriter methods 2a>+≡
 Status visit(Repeat &s) {
 rewrite(s.count);
 rewrite(s.body);
 return &s;
 }

3d <rewriter methods 2a>+≡
 Status visit(Every &s) {
 rewrite(s.body);
 rewrite(s.predicate);
 return &s;
 }

3e <rewriter methods 2a>+≡
 Status visit(Suspend &s) {
 rewrite(s.predicate);
 rewrite(s.body);
 return &s;
 }

3f <rewriter methods 2a>+≡
 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;
 }

```

```
4a <rewriter methods 2a>+≡
 Status visit(PredicatedStatement &s) {
 rewrite(s.predicate);
 rewrite(s.body);
 return &s;
 }

4b <rewriter methods 2a>+≡
 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;
 }

4c <rewriter methods 2a>+≡
 Status visit(IfThenElse& n) {
 rewrite(n.predicate);
 rewrite(n.then_part);
 rewrite(n.else_part);
 return &n;
 }

4d <rewriter methods 2a>+≡
 Status visit(Signal& s) {
 rewrite(s.body);
 return &s;
 }

4e <rewriter methods 2a>+≡
 Status visit(Var& s) {
 rewrite(s.body);
 return &s;
 }

4f <rewriter methods 2a>+≡
 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;
 }
```

- 5a *<rewriter methods 2a>+≡*  

```

Status visit(Emit& s) {
 rewrite(s.value);
 return &s;
}

```
- 5b *<rewriter methods 2a>+≡*  

```

Status visit(Assign& s) {
 rewrite(s.value);
 return &s;
}

```
- 5c *<rewriter methods 2a>+≡*  

```

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;
}

```
- 5d *<rewriter methods 2a>+≡*  

```

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;
}

```

## 1.2 Leaf Statements

These stop the recursion and return themselves;

- 5e *<rewriter methods 2a>+≡*  

```

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(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; }

```

### 1.3 Expressions

```

6a <rewriter methods 2a>+≡
 Status visit(LoadVariableExpression &e) { return &e; }
 Status visit(LoadSignalExpression &e) { return &e; }
 Status visit(LoadSignalValueExpression &e) { return &e; }
 Status visit(Literal &e) { return &e; }

6b <rewriter methods 2a>+≡
 Status visit(UnaryOp &e) {
 rewrite(e.source);
 return &e;
 }

6c <rewriter methods 2a>+≡
 Status visit(BinaryOp &e) {
 rewrite(e.source1);
 rewrite(e.source2);
 return &e;
 }

6d <rewriter methods 2a>+≡
 Status visit(FunctionCall &e) {
 for (vector<Expression*>::iterator i = e.arguments.begin() ;
 i != e.arguments.end() ; i++) {
 assert(*i);
 rewrite(*i);
 }
 return &e;
 }

6e <rewriter methods 2a>+≡
 Status visit(Delay &e) {
 rewrite(e.predicate);
 rewrite(e.count);
 return &e;
 }

```

## 2 Statement Dismantlers

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.

Once this pass is complete,

- Present and If statements have been converted to cascades of IfThenElse statements
- Await, Do watching, and Do Upto statements have been replaced with appropriate Abort statements
- Weak abort statements have been replaced with equivalent cascades of Trap statements.
- Traps with multiple traps and/or handlers have been replaced with a single, more complex handler.
- Loop Each has been replaced with Look and Abort
- Halt has been replaced with loop pause end.
- Sustain has been replaced by a loop and emit.
- Nothing has been replaced by an empty instruction sequence.

```
7 <first pass class 7>≡
 class Dismantler1 : public Rewriter {
 public:
 <first pass methods 8>
 };
```

## 2.1 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

```

```

8 (first pass methods 8)≡
 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); }

```



## 2.2 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

```

```

9 (first pass methods 8)+≡
 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);
 }

```

## 2.3 Trap

Multi-handler trap statements are transformed into a single one.

```

trap T1, T2 in
 s1
handle T1 and T2 do s2
handle not T2 do s3
end trap

trap T1, T2 in
 s1
handle T1 or T2 do
 if [T1 and T2] then s2
 ||
 if [not T2] then s3
end

```

```

10 (first pass methods 8)+≡
 Status visit(Trap &t) {
 assert(t.symbols);
 if (t.handlers.size() > 1 ||
 (t.handlers.size() >= 1 && t.symbols->size() > 1)) {

 // More than one trap or more than one handler: transform

 ParallelStatementList *psl = new ParallelStatementList();

 BuiltinTypeSymbol *boolean_type = NULL;

 for (vector<PredicatedStatement*>::const_iterator i = t.handlers.begin() ;
 i != t.handlers.end() ; i++) {
 assert(*i);
 assert((*i)->predicate);
 IfThenElse *ite = new IfThenElse((*i)->predicate, (*i)->body, NULL);
 boolean_type = dynamic_cast<BuiltinTypeSymbol*>((*i)->predicate->type);
 psl->threads.push_back(ite);
 }

 assert(boolean_type); // Should have found at least one

 Expression *newExpr = NULL;

 for (SymbolTable::const_iterator i = t.symbols->begin() ;
 i != t.symbols->end() ; i++) {
 SignalSymbol *ss = dynamic_cast<SignalSymbol*>(*i);
 assert(ss);
 assert(ss->kind == SignalSymbol::Trap);
 LoadSignalExpression *lse = new LoadSignalExpression(boolean_type, ss);
 if (newExpr)
 newExpr = new BinaryOp(boolean_type, "or", newExpr, lse);
 else
 newExpr = lse;
 }

 assert(newExpr); // Should have found at least one

```

```

 t.handlers.clear(); // Old handlers are now unneeded: should garbage collect

 t.newHandler(newExpr, ps1);
}

assert(t.handlers.size() <= 1);

return Rewriter::visit(t);
}

```

## 2.4 Weak Abort

```

weak abort trap T1 in
 b trap T2 in
when trap T3 in
 case p1 do h1 b;
 case p2 do h2 exit T3
end weak abort ||
 await
 case p1 do exit T1
 case p2 do exit T2
 end await
 end trap % T3
 handle T2 do h2
 end trap % T2
 handle T1 do h1
 end trap % T1

```

```

11 <first pass methods 8>+≡
 Trap *newTrap(SignalSymbol *&ts) {
 static unsigned int nextIndex = 0;

 Trap *result = new Trap();
 result->symbols = new SymbolTable();
 // Note: The parent of this symbol table is incorrectly NULL

 char buf[10];
 sprintf(buf, "%d", nextIndex++);
 string name = "weak_trap_" + string(buf);
 ts = new SignalSymbol(name, NULL, SignalSymbol::Trap, NULL, NULL, NULL);
 result->symbols->enter(ts);
 return result;
 }

```

```

12 <first pass methods 8>+≡
 Status visit(Abort &a) {
 if (a.is_weak) {

 SignalSymbol *innerTrap;
 Trap *inner = newTrap(innerTrap);

 StatementList *newbody = new StatementList();
 if (a.body) *newbody << a.body;
 *newbody << new Exit(innerTrap, 0);

 Await *await = new Await();

 ParallelStatementList *psl = new ParallelStatementList();
 psl->threads.push_back(newbody);
 psl->threads.push_back(await);

 inner->body = psl;

 Statement *result = inner;

 BuiltinTypeSymbol *boolean_type =
 dynamic_cast<BuiltinTypeSymbol*>(module->types->get("boolean"));
 assert(boolean_type);

 for (vector<PredicatedStatement*>::reverse_iterator i = a.cases.rbegin() ;
 i != a.cases.rend() ; i++) {
 SignalSymbol *trapSymbol;
 Trap *theNewTrap = newTrap(trapSymbol);
 theNewTrap->body = result;
 Statement *body = (*i)->body;
 if (!body) body = new Nothing();
 theNewTrap->newHandler(
 new LoadSignalExpression(boolean_type, trapSymbol), body);

 await->cases.insert(await->cases.begin(),
 new PredicatedStatement(new Exit(trapSymbol, NULL), (*i)->predicate));

 result = theNewTrap;
 }

 return transform(result);
 } else {
 // A normal Abort: recurse
 return Rewriter::visit(a);
 }
 }
}

```

## 2.5 Var

```

var v1 := e1, v2 := e2 in
 b
end var

var v1, v2 in
 v1 := e1;
 v2 := e2;
 b
end var

```

```

13 (first pass methods 8)+≡
 Status visit(Var &v) {

 bool hasInitializer = false;

 assert(v.symbols);

 for (SymbolTable::const_iterator i = v.symbols->begin() ;
 i != v.symbols->end() ; i++) {
 VariableSymbol *vs = dynamic_cast<VariableSymbol *>(*i);
 assert(vs);
 if (vs->initializer) {
 hasInitializer = true;
 break;
 }
 }

 if (hasInitializer) {
 StatementList *sl = new StatementList();

 for (SymbolTable::const_iterator i = v.symbols->begin() ;
 i != v.symbols->end() ; i++) {
 VariableSymbol *vs = dynamic_cast<VariableSymbol *>(*i);
 assert(vs);
 if (vs->initializer) {
 *sl << new Assign(vs, vs->initializer);
 vs->initializer = NULL;
 }
 }
 // Add the body of the var statement to the list
 *sl << v.body;
 v.body = sl;
 }

 rewrite(v.body);
 return &v;
 }

```

## 2.6 Do Watching and Do Upto

|     |                                                                               |  |                    |
|-----|-------------------------------------------------------------------------------|--|--------------------|
|     | <b>do</b>                                                                     |  | <b>abort</b>       |
|     | <b>b</b>                                                                      |  | <b>b</b>           |
|     | <b>watching p timeout s</b>                                                   |  | <b>when p do s</b> |
| 14a | <i>&lt;first pass methods s&gt;+≡</i>                                         |  |                    |
|     | Status visit(DoWatching &s) {                                                 |  |                    |
|     | return transform(new Abort(s.body, s.predicate, s.timeout));                  |  |                    |
|     | }                                                                             |  |                    |
|     | <b>do</b>                                                                     |  | <b>abort</b>       |
|     | <b>b</b>                                                                      |  | <b>b;</b>          |
|     | <b>upto p</b>                                                                 |  | <b>halt</b>        |
|     |                                                                               |  | <b>when p</b>      |
| 14b | <i>&lt;first pass methods s&gt;+≡</i>                                         |  |                    |
|     | Status visit(DoUpto &s) {                                                     |  |                    |
|     | return transform(new Abort(&(sl() << s.body << new Halt()), s.predicate, 0)); |  |                    |
|     | }                                                                             |  |                    |

## 2.7 Loop Each

|     |                                                                      |  |               |
|-----|----------------------------------------------------------------------|--|---------------|
|     | <b>loop</b>                                                          |  | <b>loop</b>   |
|     | <b>b</b>                                                             |  | <b>abort</b>  |
|     | <b>each p</b>                                                        |  | <b>b;</b>     |
|     |                                                                      |  | <b>halt</b>   |
|     |                                                                      |  | <b>when p</b> |
|     |                                                                      |  | <b>end</b>    |
| 14c | <i>&lt;first pass methods s&gt;+≡</i>                                |  |               |
|     | Status visit(LoopEach &s) {                                          |  |               |
|     | return transform(new Loop(new Abort(&(sl() << s.body << new Halt()), |  |               |
|     | s.predicate, 0));                                                    |  |               |
|     | }                                                                    |  |               |

## 2.8 Halt

|     |                                          |  |              |
|-----|------------------------------------------|--|--------------|
|     | <b>halt</b>                              |  | <b>loop</b>  |
|     |                                          |  | <b>pause</b> |
|     |                                          |  | <b>end</b>   |
| 14d | <i>&lt;first pass methods s&gt;+≡</i>    |  |              |
|     | Status visit(Halt &s) {                  |  |              |
|     | return transform(new Loop(new Pause())); |  |              |
|     | }                                        |  |              |

## 2.9 Sustain

```

sustain s loop
 emit s;
 pause
 end
15a <first pass methods 8>+≡
 Status visit(Sustain &s) {
 return transform(new Loop(&(sl() <<
 new Emit(s.signal, s.value) << new Pause())));
 }

```

## 2.10 Nothing

A nothing statement is replaced with an empty instruction sequence.

```

15b <first pass methods 8>+≡
 Status visit(Nothing &) {
 return transform(new StatementList());
 }

```

## 3 Dismantle.hpp and .cpp

```

15c <Dismantle.hpp 15c>≡
 #ifndef _DISMANTLE_HPP
 # define _DISMANTLE_HPP

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

 namespace Dismantle {
 using namespace IR;
 using namespace AST;

 <rewriter class 1>
 <first pass class 7>
 }
 #endif
15d <Dismantle.cpp 15d>≡
 #include <stdio.h>
 #include "Dismantle.hpp"

 namespace Dismantle {
 }

```