CEC Run Module Expander

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

Contents

1 Overview

This performs macro expansion of Esterel's `run` statement, making a copy of each module. This module, which operates on the AST, should be run before the dismantler assigns completion codes, since completion code assignment is a global operation.

Berry, in the Esterel primer, writes

> All data objects (types, functions, procedures, tasks) are global to an Esterel program. Therefore, the data declarations of the instantiated submodule are exported to the parent module. If some data objects were already declared in the parent, the parent and child declarations must be the same.

The notation x/y means "use x for y."

Things that can be renamed:

- Types

  The type in the module being instantiated is either new and is copied, the same and simply redirected, or renamed to an existing type.

- Constants

  Can be new, the same, renamed to an existing constant, or replaced with an actual constant.

- Functions

  Can be new, the same, reanamed to an existing function, or replaced with a predefined function.

- Procedures

  Can be new, the same, or renamed to an existing procedure.

- Tasks

  Like procedures.

- Signals

  Berry writes, "The signal interface declarations of the instantiated module are simply discarded, as well as the relation declarations. This means that the interface signals of the instantiated submodule must exist in the parent module with the same type. Notice that a signal declared as input in the submodule is seen as global after instantiation."

One of the first tasks is identifying the root modules. These are exactly those modules that are not instantiated by any others. This is easily checked in the IC file format: a linear search of the statements gives all the Run statements and the modules they call.

2 The Rewrite function

The rewrite function—the main entry point for the expander—does two things: determines which modules are root modules and copies each of these recursively, expanding run statements as necessary. It uses the FindRoots class to find modules that are run somewhere; those that aren't are considered roots. Then, it uses the Copier class to recursively copy each of the root modules.

3a ⟨*rewrite function declaration* 3a⟩≡

```
Modules *find_roots_and_rewrite(Modules *);
```

3b ⟨*rewrite function definition* 3b⟩≡

```
Modules *find_roots_and_rewrite(Modules *m)
{
  assert(m);
  FindRoots fr(m);
  const set<Module*> *roots = fr.find();
  if (roots->empty())
    throw IR::Error("No root nodes.  Is there a cyclic module dependency?");

  // Copy each root module in the order in which it appeared
  // in the original list of modules (preserve order)
  Modules *result = new Modules();
  for ( vector<Module*>::const_iterator i = m->modules.begin() ;
        i != m->modules.end() ; i++ )
    if ( roots->find(*i) != roots->end() ) {
      Copier copier(m, result);
      result->add(copier.copyModule(*i));
    }

  return result;
}
```

3 FindRoots

The `FindRoots` class determines which modules are root modules. These are those modules that are not "run" by any others. This is a simple depth-first search.

4a ⟨*find roots class declaration* 4a⟩≡
```
class FindRoots : public Visitor {
  Modules *modules;
public:
  set<Module*> roots;
  ⟨find roots method declarations 4b⟩
};
```

4b ⟨*find roots method declarations* 4b⟩≡
```
FindRoots(Modules *m) {
  assert(m);
  modules = m;
}

virtual ~FindRoots() {}
```

The `find` method does most of the work: it initializes the root set and recursively walks every module, removing modules that are run in others.

4c ⟨*find roots method declarations* 4b⟩+≡
```
const set<Module*> *find();
```

4d ⟨*find roots method definitions* 4d⟩≡
```
const set<Module*> *FindRoots::find()
{
  // Start the set of roots off as all modules
  for ( vector<Module*>::const_iterator i = modules->modules.begin() ;
        i != modules->modules.end() ; i++ ) {
    assert(*i);
    roots.insert(*i);
  }

  // Visit each of the modules and remove any run modules from the roots set

  for ( vector<Module*>::const_iterator i = modules->modules.begin() ;
        i != modules->modules.end() ; i++ )
    recurse(*i);

  return &roots;
}
```

The `recurse` method visits statements.

4e ⟨*find roots method declarations* 4b⟩+≡
```
void recurse(ASTNode* n) { if (n) n->welcome(*this); }
```

3.1 The Run Statement

This statement is the only one that does anything productive: modules that are run are removed from the set of roots, since a module that is run is by definition not a root.

5a ⟨*find roots method declarations* 4b⟩+≡
```
Status visit(Run &);
```

5b ⟨*find roots method definitions* 4d⟩+≡
```
Status FindRoots::visit(Run &s)
{
  if ( !(modules->module_symbols.contains(s.old_name)) )
    throw IR::Error("Attempt to run unknown module " + s.old_name);
  ModuleSymbol *ms =
    dynamic_cast<ModuleSymbol*>(modules->module_symbols.get(s.old_name));
  assert(ms);
  Module *m = ms->module;
  assert(m);
  roots.erase(m);
  return Status();
}
```

3.2 Atomic Statements

These are very simple: they stop the recursion since they do not contain any other statements.

5c ⟨*find roots method declarations* 4b⟩+≡
```
Status visit(Nothing &) { return Status(); }
Status visit(Pause &) { return Status(); }
Status visit(Halt &) { return Status(); }
Status visit(Exit &) { return Status(); }
Status visit(Emit &) { return Status(); }
Status visit(Sustain &) { return Status(); }
Status visit(Assign &) { return Status(); }
Status visit(ProcedureCall &) { return Status(); }
```

3.3 Composite Statements

These all continue the recursion on each of their child statements.

Statements with a single body are simplest:

6a ⟨*find roots method declarations* 4b⟩+≡
```
Status visit(Module &m) { recurse(m.body); return Status(); }
Status visit(Loop &s) { recurse(s.body); return Status(); }
Status visit(Repeat &s) { recurse(s.body); return Status(); }
Status visit(Signal &s) { recurse(s.body); return Status(); }
Status visit(Var &s) { recurse(s.body); return Status(); }
Status visit(LoopEach &s) { recurse(s.body); return Status(); }
Status visit(Every &s) { recurse(s.body); return Status(); }
Status visit(Suspend &s) { recurse(s.body); return Status(); }
Status visit(DoUpto &s) { recurse(s.body); return Status(); }
```

6b ⟨*find roots method declarations* 4b⟩+≡
```
Status visit(DoWatching &s) {
  recurse(s.body);
  recurse(s.timeout);
  return Status();
}
```

6c ⟨*find roots method declarations* 4b⟩+≡
```
Status visit(Trap &s) {
  recurse(s.body);
  for (vector<PredicatedStatement*>::const_iterator i = s.handlers.begin() ;
       i != s.handlers.end() ; i++ ) {
    assert(*i);
    recurse((*i)->body);
  }
  return Status();
}
```

Sequences iterate over each child.

6d ⟨*find roots method declarations* 4b⟩+≡
```
Status visit(StatementList &l) {
  for (vector<Statement*>::iterator i = l.statements.begin() ;
       i != l.statements.end() ; i++ )
    recurse(*i);
  return Status();
}
```

6e ⟨*find roots method declarations* 4b⟩+≡
```
Status visit(ParallelStatementList &l) {
  for (vector<Statement*>::iterator i = l.threads.begin() ;
       i != l.threads.end() ; i++ )
    recurse(*i);
  return Status();
}
```

IfThenElse may have child statements.

7a ⟨*find roots method declarations* 4b⟩+≡
```
Status visit(IfThenElse &s) {
  recurse(s.then_part);
  recurse(s.else_part);
  return Status();
}
```

7b ⟨*find roots method declarations* 4b⟩+≡
```
Status visit(Exec &s) {
  for (vector<TaskCall*>::iterator i = s.calls.begin() ;
       i != s.calls.end() ; i++ ) {
    assert(*i);
    recurse((*i)->body);
  }
  return Status();
}
```

3.4 Case Statements

These consist of a collection of predicated statements plus an (optional) default. The following helper function handles most of them.

7c ⟨*find roots method declarations* 4b⟩+≡
```
void visitCase(CaseStatement *s) {
  for ( vector<PredicatedStatement*>::const_iterator i = s->cases.begin() ;
        i != s->cases.end() ; i++ ) {
    assert(*i);
    recurse((*i)->body);
  }
  recurse(s->default_stmt);
}
```

7d ⟨*find roots method declarations* 4b⟩+≡
```
Status visit(Present &s) { visitCase( (CaseStatement*) &s); return Status(); }
Status visit(If &s) { visitCase( (CaseStatement*) &s); return Status(); }
Status visit(Await &s) { visitCase( (CaseStatement*) &s); return Status(); }
```

7e ⟨*find roots method declarations* 4b⟩+≡
```
Status visit(Abort &s) {
  visitCase( (CaseStatement*) &s);
  recurse(s.body);
  return Status();
}
```

4 Copier

The copier is the main workhorse: it does a recursive visit of a module, copying each AST node as it goes.

The `formalToActual` map maps from symbols in the original module to symbols in the copied module. Copying a symbol typically enters it into this map.

The `copiedSymbolTableMap` maps from symbol tables in the original module to those in the copied module. This is used to set "parent" symbol table links.

A run statement can replace constants with expressions. The `newConstantExpression` map maps from variable symbols (i.e., constants) in the old module to an object that can generate a copy of the replacement expression in the new module. If a particular variable symbol doesn't appear in this map, it should be in the `formalToActual` map.

8a ⟨*copier class declaration* 8a⟩≡

```
class Copier : public Visitor {
  map<const Symbol*, Symbol*> formalToActual;
  map<const SymbolTable*, SymbolTable*> copiedSymbolTableMap;
  map<const ConstantSymbol*, ExpressionCopier*> newConstantExpression;
  map<const Counter*, Counter*> copiedCounterMap;
  Modules *oldModules;
  Modules *newModules;
  Module *moduleBeingCopied;
  Module *newModule;
public:
  ⟨copier method declarations 8b⟩
};
```

8b ⟨*copier method declarations* 8b⟩≡

```
Copier(Modules *om, Modules *nm)
  : oldModules(om), newModules(nm), moduleBeingCopied(0), newModule(0) {
  assert(nm);
  assert(om);
}

Copier(Copier *c) {
  assert(c);
  oldModules = c->oldModules;
  newModules = c->newModules;
  newModule = c->newModule;
  moduleBeingCopied = c->moduleBeingCopied;
}

virtual ~Copier() {}
```

The `copy` method is the main one: it uses the visitor to make a copy of the node and then casts is back appropriately. Null pointers are simply returned.

9a ⟨*copier method declarations* 8b⟩+≡
```
template <class T> T* copy(T* n) {
  T* result = n ? dynamic_cast<T*>(n->welcome(*this).n) : 0;
  assert(result || !n);
  return result;
}
```

The `actualSymbol` method returns a pointer to the copy of the symbol stored in the `formalToActual` map. This assumes that a copy has been made; it's an error to ask for a symbol that hasn't been registered in the table. Note: the names of the new symbol may not match the old one because it may have been renamed.

9b ⟨*copier method declarations* 8b⟩+≡
```
template <class T> T* actualSymbol(const T* s) {
  if (!s) return 0;
  // std::cerr << "actualSymbol " << s->name << std::endl;
  map<const Symbol*, Symbol*>::const_iterator i = formalToActual.find(s);
  assert(i != formalToActual.end());        // Should be in the map
  assert((*i).second);                       // Should be non-NULL
  T* result = dynamic_cast<T*>((*i).second);
  assert(result);                            // Should be the same type
  return result;
}
```

The `newSymbolTable` method returns the copy of an old symbol table. It assumes that a copy has been made.

9c ⟨*copier method declarations* 8b⟩+≡
```
SymbolTable *newSymbolTable(const SymbolTable *s) {
  assert(s);
  map<const SymbolTable*, SymbolTable*>::const_iterator i =
     copiedSymbolTableMap.find(s);
  assert(i != copiedSymbolTableMap.end());
  SymbolTable *result = (*i).second;
  assert(result);
  return result;
}
```

9d ⟨*copier method declarations* 8b⟩+≡
```
Counter *newCounter(const Counter *c) {
  if (c) {
    assert(copiedCounterMap.find(c) != copiedCounterMap.end());
    return copiedCounterMap[c];
  }
  return 0;
}
```

4.1 Symbol Table Copier

This makes a copy of each of the symbols in the given symbol table and enters the duplicate in the `formalToActual` map. It also fixes the parent pointer using `copiedSymbolTableMap` and enters the relationship between the two symbol tables in the map.

10a ⟨*copier method declarations* 8b⟩+≡

```
void copySymbolTable(const SymbolTable *, SymbolTable *);
```

10b ⟨*copier method definitions* 10b⟩≡

```
void Copier::copySymbolTable(const SymbolTable *source, SymbolTable *dest)
{
  assert(source);
  assert(dest);

  // std::cerr << "Copying a symbol table\n";
  for ( SymbolTable::const_iterator i = source->begin() ;
        i != source->end() ; i++) {
    assert(*i);
    Symbol *news = copy(*i);
    dest->enter(news);
    formalToActual.insert( std::make_pair(*i, news) );
    // std::cerr << "Copied " << news->name << std::endl;
  }

  if (source->parent)
    dest->parent = newSymbolTable(source->parent);

  copiedSymbolTableMap.insert( std::make_pair(source, dest) );
}
```

4.2 Module

The `module` copier is only used when copying a root module.

10c ⟨*copier method declarations* 8b⟩+≡

```
Module *copyModule(Module *);
```

The order in which the symbol tables is copied is important: signals must appear after variables since they have presence variables.

11a ⟨*copier method definitions* 10b⟩+≡
```
Module *Copier::copyModule(Module *m)
{
  moduleBeingCopied = m;
  assert(m->symbol);
  ModuleSymbol *ms = new ModuleSymbol(m->symbol->name);
  Module *result = new Module(ms);
  newModule = result;
  ms->module = result;

  // std::cerr << "Copying " << m->symbol->name << std::endl;

  copySymbolTable(m->types, result->types);
  copySymbolTable(m->constants, result->constants);
  copySymbolTable(m->functions, result->functions);
  copySymbolTable(m->procedures, result->procedures);
  copySymbolTable(m->tasks, result->tasks);
  copySymbolTable(m->variables, result->variables);
  copySymbolTable(m->signals, result->signals);

  for ( vector<InputRelation*>::const_iterator i = m->relations.begin() ;
        i != m->relations.end() ; i++ ) {
    assert(*i);
    result->relations.push_back(copy(*i));
  }

  for ( vector<Counter*>::const_iterator i = m->counters.begin() ;
        i != m->counters.end() ; i++ ) {
    assert(*i);
    result->counters.push_back(*i);
    copiedCounterMap[*i] = *i;
  }

  result->body = copy(m->body);
  return result;
}
```

11b ⟨*copier method declarations* 8b⟩+≡
```
Status visit(Exclusion &e) {
  Exclusion *result = new Exclusion();
  for ( vector<SignalSymbol*>::const_iterator i = e.signals.begin() ;
        i != e.signals.end() ; i++ ) {
    assert(*i);
    result->signals.push_back(actualSymbol(*i));
  }
  return result;
}
```

12a ⟨*copier method declarations* 8b⟩+≡
```
Status visit(Implication &e) {
  return new Implication(actualSymbol(e.predicate),
                         actualSymbol(e.implication));
}
```

4.3 The Run Statement

This is the main challenge: When the copier hits a "run" statement, instead of simply copying the run statement, it makes a copy of the body of the module with the apppropriate renaming of signals and other interface objects.

12b ⟨*copier method declarations* 8b⟩+≡
```
Status visit(Run &);
```

13 ⟨*copier method definitions* 10b⟩+≡

```
Status Copier::visit(Run &r)
{
  assert(newModule);

  // std::cerr << "run " << r.new_name << " / " << r.old_name << std::endl;

  // Locate the old module by its name
  assert(oldModules);
  if ( !(oldModules->module_symbols.contains(r.old_name)) )
    throw IR::Error("Attempt to run unknown module " + r.old_name);

  ModuleSymbol *ms =
    dynamic_cast<ModuleSymbol*>(oldModules->module_symbols.get(r.old_name));
  assert(ms);
  Module *moduleToCopy = ms->module;
  assert(moduleToCopy);

  Statement *body = dynamic_cast<Statement*>(moduleToCopy->body);
  assert(body);

  Copier newcopier(this);
  newcopier.moduleBeingCopied = moduleToCopy;

  // Map the declaration symbol tables in the module being run to the
  // equivalent ones in the new module
  newcopier.copiedSymbolTableMap.insert(std::make_pair( moduleToCopy->constants,
                                                        newModule->constants));

  ⟨renamed signal rules 14⟩
  ⟨normal signal rules 15⟩

  ⟨renamed type rules 16⟩
  ⟨normal type rules 17⟩

  ⟨renamed constant rules 18⟩
  ⟨normal constant rules 19⟩

  ⟨renamed function rules 21⟩
  ⟨normal function rules 22⟩

  ⟨renamed procedure rules 23⟩
  ⟨normal procedure rules 24⟩

  ⟨renamed task rules 25⟩
  ⟨normal task rules 26⟩

  ⟨variable rules 27a⟩

  ⟨counter rules 27b⟩
```

```
    Statement *result = newcopier.copy(body);
    return result;
  }
```

4.3.1 Signals

There are three cases for each signal in the module being run: it is assigned to a new signal by renaming it with the *run* statement, it matches the name of a signal in the scope where the *run* statement appeared, or it matches the name of a signal in the enclosing scope that had been renamed by an enclosing *run* statement.

This code handles signals that are being renamed by this *run* statement. The signal renaming object provides two things: the name of the signal in the module being run and a pointer to the symbol that will replace it. This code locates the signal in the run module by name, fetches its symbol, and adds a map of this symbol in the run module to the new copy of the new signal.

14 ⟨*renamed signal rules* 14⟩≡

```
for ( vector<SignalRenaming*>::const_iterator i = r.signals.begin() ;
      i != r.signals.end() ; i++ ) {
  SignalRenaming *sr = *i;
  assert(sr);
  if ( !(moduleToCopy->signals->local_contains(sr->old_name)) )
    throw IR::Error("Attempting to rename unknown signal " + sr->old_name);
  SignalSymbol *formalSignalSymbol =
    dynamic_cast<SignalSymbol*>(moduleToCopy->signals->get(sr->old_name));
  assert(formalSignalSymbol);
  SignalSymbol *actualSignalSymbol = actualSymbol(sr->new_sig);
  newcopier.formalToActual.insert(std::make_pair(formalSignalSymbol,
                                                 actualSignalSymbol));
}
```

This code, run after renamed signals have been entered in the map, finds a matching signal for each in the interface of the run module. It looks for corresponding signals in the symbol table of signals attached to the *run* statement, which represents the scope in which the run statement appears. It skips any signal for which a mapping already exists; the renaming code above took care of it.

15 ⟨*normal signal rules* 15⟩≡

```
  assert(r.signalScope); // Should have been put there during static semantics
  // Since the run statement's symbol table is merely a pointer to a
  // table in its scope, we should have already made a new version of
  // the symbol table when we were copying the scope that encloses the
  // current run statement.
  SymbolTable *newSignalTable = newSymbolTable(r.signalScope);

/*
  for (SymbolTable::const_iterator i = r.signalScope->begin() ;
        i != r.signalScope->end() ; i++ )
        std::cerr << "symbol " << (*i)->name << '\n';
  std::cerr << "end of symbols\n";
*/

  newcopier.copiedSymbolTableMap.insert(std::make_pair(moduleToCopy->signals,
                                                       newSignalTable));

  for ( SymbolTable::const_iterator i = moduleToCopy->signals->begin() ;
        i != moduleToCopy->signals->end() ; i++ ) {
    SignalSymbol *formalSignal = dynamic_cast<SignalSymbol*>(*i);
    assert(formalSignal);
    if ( newcopier.formalToActual.find(formalSignal) ==
         newcopier.formalToActual.end() ) {
      // Haven't yet copied this symbol: locate its name in the scope of
      // the run statement
      if (!r.signalScope->contains(formalSignal->name))
        throw IR::Error("Could not find signal " + formalSignal->name +
                        " in enclosing scope (run " + r.old_name + " in " +
                        moduleBeingCopied->symbol->name + ")");
      SignalSymbol *originalActualSymbol =
          dynamic_cast<SignalSymbol*>(r.signalScope->get(formalSignal->name));
      assert(originalActualSymbol);
      // Now that we have the symbol that was used earlier, i.e., before
      // any renaming by an earlier run statement, get the renamed symbol,
      // if there is one.
      assert(formalToActual.find(originalActualSymbol) !=
             formalToActual.end());
      SignalSymbol *renamedActualSignal =
        dynamic_cast<SignalSymbol*>(formalToActual[originalActualSymbol]);
      assert(renamedActualSignal);
      newcopier.formalToActual.insert( std::make_pair(formalSignal,
                                                      renamedActualSignal) );
```

```
  }
}
```

4.3.2 Types

There are three cases for each type in the run module: the type is new and is added to the running module, the type already exists in the running module, or the type is the same in the running module, but was renamed by an enclosing *run* statement.

This code handles type renaming: for each renamed type, make sure its name is a type in the run module, then add a map from the old type symbol to the symbol in the running module.

16 ⟨*renamed type rules* 16⟩≡

```
for ( vector<TypeRenaming*>::const_iterator i = r.types.begin() ;
      i != r.types.end() ; i++ ) {
  TypeRenaming *tr = *i;
  assert(tr);
  if ( !(moduleToCopy->types->local_contains(tr->old_name)) )
    throw IR::Error("Attempting to rename unknown type " + tr->old_name);
  TypeSymbol *oldType =
    dynamic_cast<TypeSymbol*>(moduleToCopy->types->get(tr->old_name));
  assert(oldType);
  TypeSymbol *newType = actualSymbol(tr->new_type);
  newcopier.formalToActual.insert(std::make_pair(oldType, newType));
}
```

This code handles the other two cases. For each type in the run module, it first checks to see that it has not already been added, then either uses an existing identically-named type or adds it to the running module.

17 ⟨*normal type rules* 17⟩≡

```
/*
  std::cerr << "Start of symbol table" << std::endl;
  for ( SymbolTable::const_iterator i = newModule->types->begin() ;
        i != newModule->types->end() ; i++ ) {
    assert(*i);
    std::cerr << "type " << (*i)->name << std::endl;
  }
  std::cerr << "End of symbol table" << std::endl;
*/

  for ( SymbolTable::const_iterator i = moduleToCopy->types->begin() ;
        i != moduleToCopy->types->end() ; i++ ) {
    TypeSymbol *formalType = dynamic_cast<TypeSymbol*>(*i);
    assert(formalType);
    // std::cerr << "copying type " << formalType->name << std::endl;

    if ( newcopier.formalToActual.find(formalType) ==
         newcopier.formalToActual.end() ) {

      // std::cerr << "did not find it already copied" << std::endl;

      if ( moduleBeingCopied->types->local_contains(formalType->name) ) {
        // Existed already in the enclosing scope; reuse that one
        // std::cerr << "has the same name as existing type" << std::endl;
        assert(moduleBeingCopied);
        TypeSymbol *originalActualType =
           dynamic_cast<TypeSymbol*>(moduleBeingCopied->types->get(formalType->name));
        assert(originalActualType);

        // This could have been renamed earlier.  In any case, recover
        // the actual type that we had been using for it.

        /*
        std::cerr << "formalToActual map:\n";
        for (map<const Symbol*, Symbol*>::const_iterator i =
               formalToActual.begin() ; i != formalToActual.end() ; ++i) {
          const Symbol *s1 = (*i).first;
          const Symbol *s2 = (*i).second;
          std::cerr  << (*i).first << ' ' << (*i).second << ' ' <<
            s1->name << " -> " << s2->name << std::endl;
        }
        std::cerr << "Looking for " << originalActualType << std::endl;
        */

        assert(formalToActual.find(originalActualType) !=
```

```
              formalToActual.end());
        TypeSymbol *renamedActualType =
           dynamic_cast<TypeSymbol*>(formalToActual[originalActualType]);
        assert(renamedActualType);
        newcopier.formalToActual.insert( std::make_pair(formalType,
                                                renamedActualType) );

      } else {
        // Did not find in in the enclosing scope; check whether it's
        // already in the new top-level module
        if (newModule->types->local_contains(formalType->name)) {
          TypeSymbol *originalActualType =
            dynamic_cast<TypeSymbol*>(newModule->types->get(formalType->name));
          assert(originalActualType);
          newcopier.formalToActual.insert(std::make_pair( formalType,
                                                originalActualType));
        } else {
          // Did not find the type in the new module.  Copy the new type
          // into the global module we're constructing and build the link
          TypeSymbol *originalActualType = newcopier.copy(formalType);
          assert(originalActualType);
          newModule->types->enter(originalActualType);
          newcopier.formalToActual.insert( std::make_pair(formalType,
                                                originalActualType) );
        }
      }
    }
  }
```

4.3.3 Constants

Handle renamed constants: These are replaced with expressions by registering them in the newConstantExpression map

18 ⟨*renamed constant rules* 18⟩≡

```
  for ( vector<ConstantRenaming*>::const_iterator i = r.constants.begin() ;
          i != r.constants.end() ; i++ ) {
    ConstantRenaming *cr = *i;
    assert(cr);
    if ( !(moduleToCopy->constants->local_contains(cr->old_name)) )
      throw IR::Error("Attempting to rename unknown constant " + cr->old_name);
    ConstantSymbol *oldConstant =
      dynamic_cast<ConstantSymbol*>(moduleToCopy->constants->get(cr->old_name));
    assert(oldConstant);
    ExpressionCopier *ec = new ExpressionCopier(cr->new_value, this);
    newcopier.newConstantExpression.insert(std::make_pair(oldConstant, ec));
  }
```

19 ⟨*normal constant rules* 19⟩≡

```
for ( SymbolTable::const_iterator i = moduleToCopy->constants->begin() ;
        i != moduleToCopy->constants->end() ; i++ ) {
  ConstantSymbol *formalConstant = dynamic_cast<ConstantSymbol*>(*i);
  assert(formalConstant);
  // Here, we have a constant in the instantiated module.  It may be that
  // 1. The constant is new: add it to the module being built
  // 2. The constant already exists in the main module: use that one
  // 3. The constant has been renamed: do nothing; this was handled above

  if ( newcopier.newConstantExpression.find(formalConstant) ==
         newcopier.newConstantExpression.end() ) {

    // Have not already renamed this constant

    if ( moduleBeingCopied->constants->local_contains(formalConstant->name) ) {
        // Found the constant: use the existing one
      assert(moduleBeingCopied);
      ConstantSymbol *originalActualConstant =
        dynamic_cast<ConstantSymbol*>(moduleBeingCopied->constants->get(formalConstant->name));
      /* FIXME: Should verify that the initial value of the two
           constants is consistent

           module Foo:
           constant c = 10 : integer;
           run Bar
           end module

           module Bar:
           constant c = 20 : integer;
           nothing
           end module
      */

      // This constant could have already been renamed.  In any case, recover
      // the expression we had been using for it

      /*
      std::cerr << "newConstantExpression map:\n";
      for (map<const ConstantSymbol*, ExpressionCopier*>::const_iterator i =
             newConstantExpression.begin() ;
           i != newConstantExpression.end() ; ++i) {
        const ConstantSymbol *s1 = (*i).first;
        std::cerr  << (*i).first << ' ' << s1->name << std::endl;
      }
      std::cerr << "Looking for " << originalActualConstant << ' ' <<
        originalActualConstant->name << std::endl;
      */

      if (newConstantExpression.find(originalActualConstant) !=
```

```
        newConstantExpression.end()) {
      // Has been renamed to something new
      ExpressionCopier *renamedActualConstant =
        dynamic_cast<ExpressionCopier*>(newConstantExpression[originalActualConstant]);
      assert(renamedActualConstant);
      newcopier.newConstantExpression.insert( std::make_pair(formalConstant,
                                                             renamedActualConstant));
    } else {
      // Just use existing symbol
      assert(formalToActual.find(originalActualConstant) !=
             formalToActual.end());
      ConstantSymbol *renamedActualConstant =
        dynamic_cast<ConstantSymbol*>(formalToActual[originalActualConstant]);
      assert(renamedActualConstant);
      newcopier.formalToActual.insert( std::make_pair(formalConstant,
                                                      renamedActualConstant));
    }

  } else {
    // Did not find in the enclosing scope; check whether it's
    // already in the new top-level module
    if (newModule->constants->local_contains(formalConstant->name)) {
      ConstantSymbol *actualConstant =
        dynamic_cast<ConstantSymbol*>(newModule->constants->get(formalConstant->name));
      assert(actualConstant);
      newcopier.formalToActual.insert(std::make_pair(formalConstant,
                                                     actualConstant));
    } else {

      // Constant doesn't already exist in the new module: copy it
      // std::cerr << "Making a new copy of " << oldConstant->name << std::endl;
      ConstantSymbol *actualConstant = newcopier.copy(formalConstant);
      assert(actualConstant);
      newModule->constants->enter(actualConstant);

      newcopier.formalToActual.insert( std::make_pair(formalConstant,
                                                      actualConstant) );

      /* FIXME: This won't generate re-parsable code if the
         constant's name was hidden by a local variable decalaration, e.g.,

         module Foo:
         var c : integer in
           run Bar
         end
         end module

         module Bar:
         constant c : integer;
         nothing
```

```
            end module

            Berry's iclc doesn't worry about this because it simply numbers
            everything and stores the names separately.
         */
       }
     }
   }
 }
```

4.3.4 Functions

21 ⟨*renamed function rules* 21⟩≡

```
/* FIXME: Renaming built-in functions, such as +, will generate
   strange output,

   e.g., c := +(a,b)
 */

for ( vector<FunctionRenaming*>::const_iterator i = r.functions.begin() ;
        i != r.functions.end() ; i++ ) {
  FunctionRenaming *fr = *i;
  assert(fr);
  if ( !(moduleToCopy->functions->local_contains(fr->old_name)) )
    throw IR::Error("Attempting to rename unknown function " + fr->old_name);
  FunctionSymbol *oldFunction =
    dynamic_cast<FunctionSymbol*>(moduleToCopy->functions->get(fr->old_name));
  assert(oldFunction);
  FunctionSymbol *newFunction = actualSymbol(fr->new_func);
  newcopier.formalToActual.insert(std::make_pair(oldFunction, newFunction));
}
```

22 ⟨*normal function rules* 22⟩≡

```
for ( SymbolTable::const_iterator i = moduleToCopy->functions->begin() ;
        i != moduleToCopy->functions->end() ; i++ ) {
  FunctionSymbol *formalFunction = dynamic_cast<FunctionSymbol*>(*i);
  assert(formalFunction);
  // Here, we have a function in the instantiated module.  It may be that
  // 1. The function is new: add it to the module being built
  // 2. The function already exists in the main module: use that one
  // 3. The function has been renamed: do nothing; this was handled above

  if ( newcopier.formalToActual.find(formalFunction) ==
         newcopier.formalToActual.end() ) {

    // Haven't already renamed this function

    if ( moduleBeingCopied->functions->local_contains(formalFunction->name) ) {
        // Found the function: use the existing one
        FunctionSymbol *originalActualFunction =
         dynamic_cast<FunctionSymbol*>(moduleBeingCopied->functions->get(formalFunction->name)
        assert(originalActualFunction);

        // FIXME: Should check that the parameter types are consistent
        assert(formalToActual.find(originalActualFunction) !=
               formalToActual.end());
        FunctionSymbol *renamedActualFunction =
         dynamic_cast<FunctionSymbol*>(formalToActual[originalActualFunction]);
        assert(renamedActualFunction);
        newcopier.formalToActual.insert( std::make_pair( formalFunction,
                                                         renamedActualFunction));

    } else {
      // Function isn't in the enclosing scope; see if it's already in
      // the new top-level module
      if (newModule->functions->local_contains(formalFunction->name)) {
        FunctionSymbol *originalActualFunction =
          dynamic_cast<FunctionSymbol*>(newModule->functions->get(formalFunction->name));
        assert(originalActualFunction);
        newcopier.formalToActual.insert( std::make_pair( formalFunction,
                                                         originalActualFunction ));

      } else {
        // Function doesn't already exist in the new module: copy it
        FunctionSymbol *originalActualFunction =
          newcopier.copy(formalFunction);
        assert(originalActualFunction);
        newModule->functions->enter(originalActualFunction);
        newcopier.formalToActual.insert( std::make_pair(formalFunction,
                                                        originalActualFunction) );
      }
    }
```

```
  }
}
```

4.3.5 Procedures

Renamed procedures are much like renamed constants or functions: map each old procedure symbol to the new one.

23 ⟨*renamed procedure rules* 23⟩≡

```
for ( vector<ProcedureRenaming*>::const_iterator i = r.procedures.begin() ;
        i != r.procedures.end() ; i++ ) {
  ProcedureRenaming *pr = *i;
  assert(pr);
  if ( !(moduleToCopy->procedures->local_contains(pr->old_name)) )
    throw IR::Error("Attempting to rename unknown procedure " + pr->old_name);
  ProcedureSymbol *oldProcedure =
    dynamic_cast<ProcedureSymbol*>(moduleToCopy->procedures->get(pr->old_name));
  assert(oldProcedure);
  ProcedureSymbol *newProcedure = actualSymbol(pr->new_proc);
  newcopier.formalToActual.insert(std::make_pair(oldProcedure, newProcedure));
}
```

Like functions, there are three cases for procedures that are not being renamed:

1. The procedure in the module being run is new; add it to the module being built
2. The procedure already exists in the main module; use it
3. The procedure is being renamed. This was handled above.

24 ⟨*normal procedure rules* 24⟩≡

```
for ( SymbolTable::const_iterator i = moduleToCopy->procedures->begin() ;
        i != moduleToCopy->procedures->end() ; i++ ) {
  ProcedureSymbol *formalProcedure = dynamic_cast<ProcedureSymbol*>(*i);
  assert(formalProcedure);

  if ( newcopier.formalToActual.find(formalProcedure) ==
       newcopier.formalToActual.end() ) {

    // Haven't already renamed this procedure

    if ( moduleBeingCopied->procedures->local_contains(formalProcedure->name) ) {
        // Found the procedure: use the existing one
      ProcedureSymbol *originalActualProcedure =
        dynamic_cast<ProcedureSymbol*>(moduleBeingCopied->procedures->get(formalProcedure->nam

        // FIXME: Verify the parameter count and types are consistent
      assert(originalActualProcedure);

      assert(formalToActual.find(originalActualProcedure) !=
             formalToActual.end());
      ProcedureSymbol *renamedActualProcedure =
        dynamic_cast<ProcedureSymbol*>(formalToActual[originalActualProcedure]);
      assert(renamedActualProcedure);
      newcopier.formalToActual.insert(std::make_pair( formalProcedure,
                                                      renamedActualProcedure));


    } else {
        // Procedure isn't already in scope: check to see whether it
        // already exists (e.g., whether it was copied by another module)

        if (newModule->procedures->local_contains(formalProcedure->name)) {
          // Procedure already exists; use that one
          ProcedureSymbol *originalActualProcedure =
            dynamic_cast<ProcedureSymbol*>(newModule->procedures->get(formalProcedure->name));
          assert(originalActualProcedure);
          newcopier.formalToActual.insert(std::make_pair( formalProcedure,
                                                          originalActualProcedure ));
        } else {
```

```
          // Procedure doesn't already exist in the new module: copy it
          ProcedureSymbol *originalActualProcedure =
            newcopier.copy(formalProcedure);
          assert(originalActualProcedure);
          newModule->procedures->enter(originalActualProcedure);
          newcopier.formalToActual.insert( std::make_pair(formalProcedure,
                                                 originalActualProcedure) );
        }
      }
    }
  }
```

4.3.6 Tasks

Tasks are essentially identical to procedures and handled similarly.

25 ⟨*renamed task rules* 25⟩≡

```
  for ( vector<ProcedureRenaming*>::const_iterator i = r.tasks.begin() ;
          i != r.tasks.end() ; i++ ) {
    ProcedureRenaming *pr = *i;
    assert(pr);
    if ( !(moduleToCopy->tasks->local_contains(pr->old_name)) )
      throw IR::Error("Attempting to rename unknown task " + pr->old_name);
    TaskSymbol *oldTask =
      dynamic_cast<TaskSymbol*>(moduleToCopy->tasks->get(pr->old_name));
    assert(oldTask);
    TaskSymbol *newTaskSymbol = dynamic_cast<TaskSymbol*>(pr->new_proc);
    assert(newTaskSymbol);
    TaskSymbol *newTask = actualSymbol(newTaskSymbol);
    newcopier.formalToActual.insert(std::make_pair(oldTask, newTask));
  }
```

26 ⟨*normal task rules* 26⟩≡

```
for ( SymbolTable::const_iterator i = moduleToCopy->tasks->begin() ;
        i != moduleToCopy->tasks->end() ; i++ ) {
  TaskSymbol *formalTask = dynamic_cast<TaskSymbol*>(*i);
  assert(formalTask);

  if ( newcopier.formalToActual.find(formalTask) ==
       newcopier.formalToActual.end() ) {

    // Haven't already renamed this task

    if ( moduleBeingCopied->tasks->local_contains(formalTask->name) ) {
        // Found the task: use the existing one
        TaskSymbol *originalActualTask =
          dynamic_cast<TaskSymbol*>(moduleBeingCopied->tasks->get(formalTask->name));
        assert(originalActualTask);

        // FIXME: Verify the parameter count and types are consistent

        assert(formalToActual.find(originalActualTask) !=
               formalToActual.end());
        TaskSymbol *renamedActualTask =
          dynamic_cast<TaskSymbol*>(formalToActual[originalActualTask]);
        assert(renamedActualTask);
        newcopier.formalToActual.insert( std::make_pair( formalTask,
                                                         renamedActualTask));

    } else {
        // Task doesn't already exist in the new module: copy it
        TaskSymbol *originalActualTask = newcopier.copy(formalTask);
        assert(originalActualTask);
        newModule->tasks->enter(originalActualTask);
        newcopier.formalToActual.insert( std::make_pair(formalTask,
                                                        originalActualTask) );
    }

  }
}
```

4.3.7 Variables

Although normal variables aren't visible in the run module, variables that are owned by the module (e.g., signal and trap presence and value variables) need to be copied. The main trick is ensuring the names don't collide: this is done by prepending the module's new name to the existing variable name and adding a suffix if necessary.

27a ⟨*variable rules* 27a⟩≡

```
for ( SymbolTable::const_iterator i = moduleToCopy->variables->begin() ;
        i != moduleToCopy->variables->end() ; i++ ) {
  VariableSymbol *oldVariable = dynamic_cast<VariableSymbol*>(*i);
  assert(oldVariable);
  string baseName = r.new_name + "_" + oldVariable->name;
  string newName = baseName;
  int next = 1;
  while (newModule->variables->local_contains(newName)) {
    char buf[10];
    sprintf(buf, "%d", next++);
    newName = baseName + '_' + buf;
  }
  TypeSymbol *newType = newcopier.actualSymbol(oldVariable->type);
  Expression *newInitializer = newcopier.copy(oldVariable->initializer);
  VariableSymbol *newVariable =
     new VariableSymbol(newName, newType, newInitializer);
  newModule->variables->enter(newVariable);
  newcopier.formalToActual.insert( std::make_pair(oldVariable, newVariable));
//  std::cerr << "copying " << oldVariable->name << " to " << newName << std::endl;
}

copiedSymbolTableMap.insert( std::make_pair(moduleToCopy->variables,
                                            newModule->variables) );
```

4.3.8 Counters

Counters are implicit in Esterel but play an important role in counted delays (e.g., `await 5 A`) and *repeat* statements. They are simply hoisted from instantiated modules into the topmost one.

27b ⟨*counter rules* 27b⟩≡

```
for ( vector<Counter*>::const_iterator i = moduleToCopy->counters.begin() ;
        i != moduleToCopy->counters.end() ; i++ ) {
  Counter *newCounter = new Counter();
  newModule->counters.push_back(newCounter);
  newcopier.copiedCounterMap[*i] = newCounter;
}
```

4.4 Symbols

These simply copy their names and data; the `copySymbolTable` method is responsible for entering them in the map.

28a ⟨*copier method declarations* 8b⟩+≡
```
Status visit(TypeSymbol &s) { return new TypeSymbol(s.name); }

Status visit(BuiltinTypeSymbol &s) { return new BuiltinTypeSymbol(s.name); }

Status visit(ConstantSymbol &s) {
  return new ConstantSymbol(s.name, actualSymbol(s.type), copy(s.initializer));
}

Status visit(BuiltinConstantSymbol &s) {
  return new BuiltinConstantSymbol(s.name, actualSymbol(s.type),
                                   copy(s.initializer));
}
```

28b ⟨*copier method declarations* 8b⟩+≡
```
Status visit(FunctionSymbol&);
Status visit(BuiltinFunctionSymbol&);
Status visit(ProcedureSymbol&);
Status visit(TaskSymbol&);
```

28c ⟨*copier method definitions* 10b⟩+≡
```
Status Copier::visit(FunctionSymbol &s)
{
  FunctionSymbol *result = new FunctionSymbol(s.name);
  result->result = actualSymbol(s.result);
  for ( vector<TypeSymbol*>::const_iterator i = s.arguments.begin() ;
        i != s.arguments.end() ; i++ ) {
    assert(*i);
    result->arguments.push_back(actualSymbol(*i));
  }
  return result;
}

Status Copier::visit(BuiltinFunctionSymbol &s)
{
  BuiltinFunctionSymbol *result = new BuiltinFunctionSymbol(s.name);
  result->result = actualSymbol(s.result);
  for ( vector<TypeSymbol*>::const_iterator i = s.arguments.begin() ;
        i != s.arguments.end() ; i++ ) {
    assert(*i);
    result->arguments.push_back(actualSymbol(*i));
  }
  return result;
}
```

29a ⟨*copier method definitions* 10b⟩+≡

```
Status Copier::visit(ProcedureSymbol &s)
{
  ProcedureSymbol *result = new ProcedureSymbol(s.name);
  for ( vector<TypeSymbol*>::const_iterator i = s.reference_arguments.begin() ;
        i != s.reference_arguments.end() ; i++ ) {
    assert(*i);
    result->reference_arguments.push_back(actualSymbol(*i));
  }
  for ( vector<TypeSymbol*>::const_iterator i = s.value_arguments.begin() ;
        i != s.value_arguments.end() ; i++ ) {
    assert(*i);
    result->value_arguments.push_back(actualSymbol(*i));
  }
  return result;
}

Status Copier::visit(TaskSymbol &s)
{
  TaskSymbol *result = new TaskSymbol(s.name);
  for ( vector<TypeSymbol*>::const_iterator i = s.reference_arguments.begin() ;
        i != s.reference_arguments.end() ; i++ ) {
    assert(*i);
    result->reference_arguments.push_back(actualSymbol(*i));
  }
  for ( vector<TypeSymbol*>::const_iterator i = s.value_arguments.begin() ;
        i != s.value_arguments.end() ; i++ ) {
    assert(*i);
    result->value_arguments.push_back(actualSymbol(*i));
  }
  return result;
}
```

29b ⟨*copier method declarations* 8b⟩+≡

```
Status visit(SignalSymbol &s)
{
  // std::cerr << "Copy signal " << s.name << std::endl;
  return
    new SignalSymbol( s.name, actualSymbol(s.type), (SignalSymbol::kinds) s.kind,
                      actualSymbol(s.combine), copy(s.initializer), actualSymbol(s.reincarnation) );
}

Status visit(BuiltinSignalSymbol &s)
{
  return
    new BuiltinSignalSymbol( s.name, actualSymbol(s.type),
                             (SignalSymbol::kinds) s.kind,
                             actualSymbol(s.combine));
}
```

30a ⟨*copier method declarations* 8b⟩+≡
```
Status visit(VariableSymbol &s) {
  return new VariableSymbol(s.name, actualSymbol(s.type), copy(s.initializer));
}
```

4.5 Atomic Statements

These simply return a new copy of themselves.

30b ⟨*copier method declarations* 8b⟩+≡
```
Status visit(Nothing &) { return new Nothing(); }
Status visit(Pause &) { return new Pause(); }
Status visit(Halt &) { return new Halt(); }
```

The `emit` and `sustain` statements have a signal and optional expression.

30c ⟨*copier method declarations* 8b⟩+≡
```
Status visit(Emit &e) {
  return new Emit(actualSymbol(e.signal), copy(e.value));
}

Status visit(Sustain &s) {
  return new Sustain(actualSymbol(s.signal), copy(s.value));
}
```

The assign statement has a variable symbol and an expression.

30d ⟨*copier method declarations* 8b⟩+≡
```
Status visit(Assign &a) {
  return new Assign(actualSymbol(a.variable), copy(a.value));
}
```

30e ⟨*copier method declarations* 8b⟩+≡
```
Status visit(Exit &e) { return new Exit(actualSymbol(e.trap), copy(e.value)); }
```

30f ⟨*copier method declarations* 8b⟩+≡
```
Status visit(ProcedureCall &);
```

30g ⟨*copier method definitions* 10b⟩+≡
```
Status Copier::visit(ProcedureCall &c)
{
  ProcedureCall *result = new ProcedureCall(actualSymbol(c.procedure));
  for ( vector<VariableSymbol*>::const_iterator i = c.reference_args.begin() ;
        i != c.reference_args.end() ; i++ ) {
    assert(*i);
    result->reference_args.push_back(actualSymbol(*i));
  }
  for ( vector<Expression*>::const_iterator i = c.value_args.begin() ;
        i != c.value_args.end() ; i++ ) {
    assert(*i);
    result->value_args.push_back(copy(*i));
  }
  return result;
}
```

4.6 Statement Lists

These make copies of all statements under their control.

31a ⟨*copier method declarations* 8b⟩+≡
```
Status visit(StatementList&);
Status visit(ParallelStatementList&);
```

31b ⟨*copier method definitions* 10b⟩+≡
```
Status Copier::visit(StatementList &l)
{
  StatementList *result = new StatementList();
  for (vector<Statement*>::iterator i = l.statements.begin() ;
       i != l.statements.end() ; i++ ) {
    *result << copy(*i);
  }
  return result;
}

Status Copier::visit(ParallelStatementList &l)
{
  ParallelStatementList *result = new ParallelStatementList();
  for (vector<Statement*>::iterator i = l.threads.begin() ;
       i != l.threads.end() ; i++ ) {
    result->threads.push_back(copy(*i));
  }
  return result;
}
```

4.7 Composite Statements

31c ⟨*copier method declarations* 8b⟩+≡
```
Status visit(IfThenElse &s) {
  return new IfThenElse(copy(s.predicate), copy(s.then_part),
                        copy(s.else_part));
}
```

31d ⟨*copier method declarations* 8b⟩+≡
```
Status visit(Loop &s) { return new Loop(copy(s.body)); }
```

31e ⟨*copier method declarations* 8b⟩+≡
```
Status visit(Repeat &s) {
  return new Repeat(copy(s.body), copy(s.count), s.is_positive,
                    newCounter(s.counter));
}
```

31f ⟨*copier method declarations* 8b⟩+≡
```
Status visit(LoopEach &s) {
  return new LoopEach( copy(s.body), copy(s.predicate));
}
```

32a ⟨*copier method declarations* 8b⟩+≡
```
Status visit(Every &s) {
  return new Every( copy(s.body), copy(s.predicate));
}
```

32b ⟨*copier method declarations* 8b⟩+≡
```
Status visit(Suspend &s) {
  return new Suspend( copy(s.body), copy(s.predicate));
}
```

32c ⟨*copier method declarations* 8b⟩+≡
```
Status visit(DoWatching &s) {
  return new DoWatching( copy(s.body), copy(s.predicate), copy(s.timeout));
}
```

32d ⟨*copier method declarations* 8b⟩+≡
```
Status visit(DoUpto &s) {
  return new DoUpto( copy(s.body), copy(s.predicate));
}
```

32e ⟨*copier method declarations* 8b⟩+≡
```
Status visit(Exec &);
```

32f ⟨*copier method definitions* 10b⟩+≡
```
Status Copier::visit(Exec &e)
{
  Exec *result = new Exec();
  for ( vector<TaskCall*>::const_iterator j = e.calls.begin() ;
        j != e.calls.end() ; j++ ) {
    TaskCall *c = *j;
    assert(c);
    TaskSymbol *ts = dynamic_cast<TaskSymbol*>(c->procedure);
    assert(ts);
    TaskCall *newCall = new TaskCall(actualSymbol(ts));
    newCall->signal = actualSymbol(c->signal);
    newCall->body = copy(c->body);
    for ( vector<VariableSymbol*>::const_iterator i = c->reference_args.begin() ;
          i != c->reference_args.end() ; i++ ) {
      assert(*i);
      newCall->reference_args.push_back(actualSymbol(*i));
    }
    for ( vector<Expression*>::const_iterator i = c->value_args.begin() ;
          i != c->value_args.end() ; i++ ) {
      assert(*i);
      newCall->value_args.push_back(copy(*i));
    }
    result->calls.push_back(newCall);
  }
  return result;
}
```

4.8 Case Statements

33a ⟨*copier method declarations* 8b⟩+≡

```
void copyCases(const CaseStatement &, CaseStatement *);
```

33b ⟨*copier method definitions* 10b⟩+≡

```
void Copier::copyCases(const CaseStatement &source, CaseStatement *dest)
{
  assert(dest);
  for ( vector<PredicatedStatement*>::const_iterator i = source.cases.begin() ;
        i != source.cases.end() ; i++) {
    assert(*i);
    dest->newCase(copy((*i)->body), copy((*i)->predicate));
  }

  dest->default_stmt = copy(source.default_stmt);
}
```

33c ⟨*copier method declarations* 8b⟩+≡

```
Status visit(Abort &s) {
  Abort *result = new Abort(copy(s.body), s.is_weak);
  copyCases(s, result);
  return result;
}
```

33d ⟨*copier method declarations* 8b⟩+≡

```
Status visit(Await &s) {
  Await *result = new Await();
  copyCases(s, result);
  return result;
}
```

33e ⟨*copier method declarations* 8b⟩+≡

```
Status visit(Present &s) {
  Present *result = new Present();
  copyCases(s, result);
  return result;
}

Status visit(If &s) {
  If *result = new If();
  copyCases(s, result);
  return result;
}
```

4.9 Scope Statements

34a ⟨*copier method declarations* 8b⟩+≡
```
Status visit(Var &s) {
  Var *result = new Var();
  result->symbols = new SymbolTable();
  copySymbolTable(s.symbols, result->symbols);
  result->body = copy(s.body);
  return result;
}
```

34b ⟨*copier method declarations* 8b⟩+≡
```
Status visit(Signal &s) {
  Signal *result = new Signal();
  result->symbols = new SymbolTable();
  copySymbolTable(s.symbols, result->symbols);
  result->body = copy(s.body);
  return result;
}
```

34c ⟨*copier method declarations* 8b⟩+≡
```
Status visit(Trap &s) {
  Trap *result = new Trap();
  result->symbols = new SymbolTable();
  copySymbolTable(s.symbols, result->symbols);
  result->body = copy(s.body);
  for ( vector<PredicatedStatement *>::const_iterator i = s.handlers.begin() ;
        i != s.handlers.end() ; i++ ) {
    assert(*i);
    result->newHandler( copy((*i)->predicate), copy((*i)->body) );
  }
  return result;
}
```

4.10 Expressions

These are all quite simple. The one trick is that the type associated with each expression is usually initialized from the expression's operands.

34d ⟨*copier method declarations* 8b⟩+≡
```
Status visit(Literal &l) { return new Literal(l.value, actualSymbol(l.type)); }
```

When a module is `run` with a literal substituted for a constant, e.g., `run MyModule [constant 3.14 / pi]` , instances of constant references (actually, load variable expressions) need to be replaced with an appropriate copy of the expression. A `CopyExpression` object records the information for doing this.

35a ⟨*expression copier class declaration* 35a⟩≡
```
class ExpressionCopier {
  Expression *theExpr;
  Copier *theCopier;
public:
  ExpressionCopier(Expression*, Copier*);
  Expression *copy();
};
```

35b ⟨*expression copier method definitions* 35b⟩≡
```
ExpressionCopier::ExpressionCopier(Expression *e, Copier *c)
  : theExpr(e), theCopier(c) { assert(e); assert(c); }

Expression *ExpressionCopier::copy() { return theCopier->copy(theExpr); }
```

35c ⟨*copier method declarations* 8b⟩+≡
```
Status visit(LoadVariableExpression &e) {
  // std::cerr << "copying variable load " << e.variable->name << std::endl;
  ConstantSymbol *cs = dynamic_cast<ConstantSymbol*>(e.variable);
  if (cs) {
    map<const ConstantSymbol*, ExpressionCopier*>::iterator i =
        newConstantExpression.find(cs);
    if (i != newConstantExpression.end())
      return (*i).second->copy();
  }
  return new LoadVariableExpression(actualSymbol(e.variable));
}
```

35d ⟨*copier method declarations* 8b⟩+≡
```
Status visit(LoadSignalExpression &e) {
  return new LoadSignalExpression(e.type, actualSymbol(e.signal));
}

Status visit(LoadSignalValueExpression &e) {
  return new LoadSignalValueExpression(actualSymbol(e.signal));
}
```

35e ⟨*copier method declarations* 8b⟩+≡
```
Status visit(Delay &d) {
  return new Delay(actualSymbol(d.type), copy(d.predicate),
                   copy(d.count), d.is_immediate, newCounter(d.counter));
}
```

36a ⟨*copier method declarations* 8b⟩+≡
```
Status visit(UnaryOp &o) {
  return new UnaryOp(actualSymbol(o.type), o.op, copy(o.source));
}

Status visit(BinaryOp &o) {
  return new BinaryOp(actualSymbol(o.type), o.op,
                      copy(o.source1), copy(o.source2));
}
```

36b ⟨*copier method declarations* 8b⟩+≡
```
Status visit(FunctionCall &);
```

36c ⟨*copier method definitions* 10b⟩+≡
```
Status Copier::visit(FunctionCall &c)
{
  FunctionCall *result = new FunctionCall(actualSymbol(c.callee));
  for ( vector<Expression*>::const_iterator i = c.arguments.begin() ;
        i != c.arguments.end() ; i++ ) {
    assert(*i);
    result->arguments.push_back(copy(*i));
  }
  return result;
}
```

5 ExpandModules.hpp and .cpp

37a ⟨*ExpandModules.hpp* 37a⟩≡

```
#ifndef _DISMANTLE_HPP
#  define _DISMANTLE_HPP

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

namespace ExpandModules {
  using namespace IR;
  using namespace AST;
  using std::set;
  using std::map;

  ⟨find roots class declaration 4a⟩
  class Copier;
  ⟨expression copier class declaration 35a⟩
  ⟨copier class declaration 8a⟩
  ⟨rewrite function declaration 3a⟩
}
#endif
```

37b ⟨*ExpandModules.cpp* 37b⟩≡

```
#include <stdio.h>
#include "ExpandModules.hpp"

namespace ExpandModules {
  using namespace IR;
  using namespace AST;
  ⟨find roots method definitions 4d⟩
  ⟨copier method definitions 10b⟩
  ⟨expression copier method definitions 35b⟩
  ⟨rewrite function definition 3b⟩
}
```

6 Notes on V5's iclc

The Run statement has an interesting field in the renamings section defined as "the index of the last signal declared at the level of the instruction expanding the module's code."

An example:

```
module Test_Run3:
input A;
output B;

run Foo;
signal C,
       D,
       E in
  pause;
  pause
end signal

end module

module Foo:

nothing

end module
```

```
signals: 5
0: input: A 1 pure: bool: 0 0 previous: - %lc: 2 7 0%
1: output: B 2 pure: previous: 0 %lc: 3 8 0%
2: local: C pure: previous: 1 %lc: 6 8 0%
3: local: D pure: previous: 2 %lc: 7 8 0%
4: local: E pure: previous: 3 %lc: 8 8 0%
end:

statements: 7
0: Return: 0 %lc: 13 1 0%
1: Run: Foo [ 1] (2) <0> %lc: 5 1 0%
2: Sigscope: [
   2 %lc: 6 8 0%,
   3 %lc: 7 8 0%,
   4 %lc: 8 8 0%
] {3} (4)
3: Endscope: [0] (0) <0> %lc: 6 1 0%
```

The index is 1, since only A and B are in scope at the run Foo statement.

Signals above these aren't visible to the module being run, but it isn't a simple less-than relationship because of "holes" in the namespace.

```
module Test_Run4:
signal A, B, C in
  nothing
end signal;
signal D, E in
  run Foo
end signal;
signal F, G in
  pause
end signal
end module

module Foo:
inputoutput D;
emit D
end module
```

```
signals: 7
0: local: A pure: previous: - %lc: 3 8 0%
1: local: B pure: previous: 0 %lc: 4 8 0%
2: local: C pure: previous: 1 %lc: 5 8 0%
3: local: D pure: previous: - %lc: 8 8 0%
4: local: E pure: previous: 3 %lc: 9 8 0%
5: local: F pure: previous: - %lc: 12 8 0%
6: local: G pure: previous: 5 %lc: 13 8 0%
end:

6: Run: Foo [ 4] (7) <5> %lc: 10 3 0%
```

The index here is 4 (corresonding to E), but 0-2 aren't visible.

The Instance table in the IC file format is trivial for the IC format (each module has exactly one instance, itself), but describes the instantiation hierarchy in an LC file.

Constant renaming can replace constant names with (trivial) constant expressions. This substitution appears to replace every instance of the constant with the appropriate expression.