

ESUIF: An Open Esterel Compiler

Stephen A. Edwards

Department of Computer Science

Columbia University

www.cs.columbia.edu/~sedwards

Not Another One...

My research agenda is to push Esterel compilation technology further.

We still don't have a technique that builds fast code for large programs.

No decent Esterel compiler available in source form.

Brief History of Esterel Compilers

Automata-based

V1, V2, V3 (INRIA/CMA) [Berry, Gonthier 1992]

Still the best for small programs with few states

Does not scale

Netlist-based

V4, V5 (INRIA/CMA)

Scales very nicely

Produces code that runs hundreds of times slower for sequential programs

Only executables available (www.esterel.org)

Brief History of Esterel Compilers

Control-flow-graph based

My work: EC [DAC 2000, TransCAD 2002]

Produces very efficient code for acyclic programs only

Discrete-event based

SAXO-RT [Weil et al. 2000]

Produces very efficient code for acyclic programs only

Being improved at Esterel Technologies?

Both proprietary; unlikely to be released.

Neither currently copes with statically cyclic programs.

ESUIF

New, open-source compiler being developed at Columbia

Based on SUIF 2 system from Stanford University

Much more modular: implemented as many little passes

Common database represents program throughout

SUIF 2 Database

Main component of the SUIF 2 system

User-customizable object-oriented database

Written in C++

Not highly efficient, but very flexible

SUIF 2 Database

Database schema written in their own “hoof” format

C++ implementation automatically generated

```
concrete MyClass {  
    int x;  
}  
⇒  
class MyClass : public SuifObject  
{  
    public:  
        int get_x();  
        void set_x(int the_value);  
        ~MyClass();  
        void print(...);  
        static const Lstring  
            get_class_name();  
}
```

Three Intermediate Representations

AST-like representation from front end

Primitives: abort, emit, present, suspend, etc.

Lower-level “C-like” representation

Primitives: if-then-else, try, resume, parallel, etc.

C code

Primitives: if, goto, expressions

SUIF 2 includes a complete C schema

My New Intermediate Representation

Intermediate Representation Goals

Linear, textual, imperative style fits the SUIF 2 philosophy

Gonthier's IC format used in V3–V5 is graph-based and difficult to visualize. Analysis requires depth-first search.

Straightforward translation into C code; simple semantics

IC format requires complicated depth-first search to linearize. Handling of “completion codes” is subtle.

Compound statements express traps, preemption, and concurrency

Tree structure present in IC, but must be rediscovered.

Intermediate Representation

var := expr

if (expr) { stmts } else { stmts }

Label:

goto Label

break n

continue

try { stmts } catch 2 { stmts } ...

resume { stmts } catch 1 { stmts } ...

parallel { resumes } catch 1 { stmts } ...

fork Label1, Label2, ...

join

Intermediate Representation

var := expr

if (expr) { stmts } else { stmts }

Label:

goto Label

Self-explanatory

Signals represented as variables.

Restrictions on where a goto may branch.

Intermediate Representation

`break n`

`continue`

`try { stmts } catch 2 { stmts } ...`

`resume { stmts } catch 1 { stmts } ...`

`parallel { resumes } catch 1 { stmts } ...`

Numerically-encoded “exceptions”

Based on Esterel’s completion codes

0=terminate 1=pause 2,3,...=exit

Implementing Exceptions

```
trap T1 in          try {
  exit T1           break 2          goto Catch2;
                                     goto Catch0;
handle T1          } catch 2 {      Catch2:
do                c := 1           c = 1;
  c := 1          }                Catch0:
end
```

`try` becomes a few labels.

`break` becomes a `goto`.

Resume/Continue

```
abort      resume {
                                goto E
                                C: switch (s) {
                                    case 0: goto St0;
                                    case 1: goto St1;
                                }
                                E: s = 0; goto Ca1; St0:
                                s = 1; goto Ca1; St1:
                                goto Ca0;
                                Ca1:
                                so = 0; goto Calo; St0o:
                                if (!A) goto C;
                                Ca0:
                                }
                                } catch 1 {
                                    break 1
                                }
when A     if (!A)
           continue
           }
```

`resume` becomes a multi-way branch plus some labels.

`continue` sends control to the multi-way branch.

Resume/Continue

First cycle:

```
goto E
C: switch (s) {
  case 0: goto St0;
  case 1: goto St1;
}
E: s = 0; goto Ca1; St0:
  s = 1; goto Ca1; St1:
  goto Ca0;
Ca1:
  so = 0; goto Ca0;
St0o: if (!A) goto C;
Ca0:
```

Second cycle:

```
goto E
C: switch (s) {
  case 0: goto St0;
  case 1: goto St1;
}
E: s = 0; goto Ca1; St0:
  s = 1; goto Ca1; St1:
  goto Ca0;
Ca1:
  so = 0; goto Ca0;
St0o: if (!A) goto C;
Ca0:
```

Parallel and Exit

```
trap T1 in  
  trap T2 in
```

```
    exit T1
```

```
  ||
```

```
    exit T2
```

```
  handle T2 do emit B end
```

```
  handle T1 do emit A end
```

```
try {  
  try {  
    parallel {  
      resume {  
        break 3 }  
      resume {  
        break 2 }  
      } catch 1 {  
        break 1; continue  
      } catch 2 { B := 1 }  
    } catch 3 { A := 1 }
```

Parallel

pause;

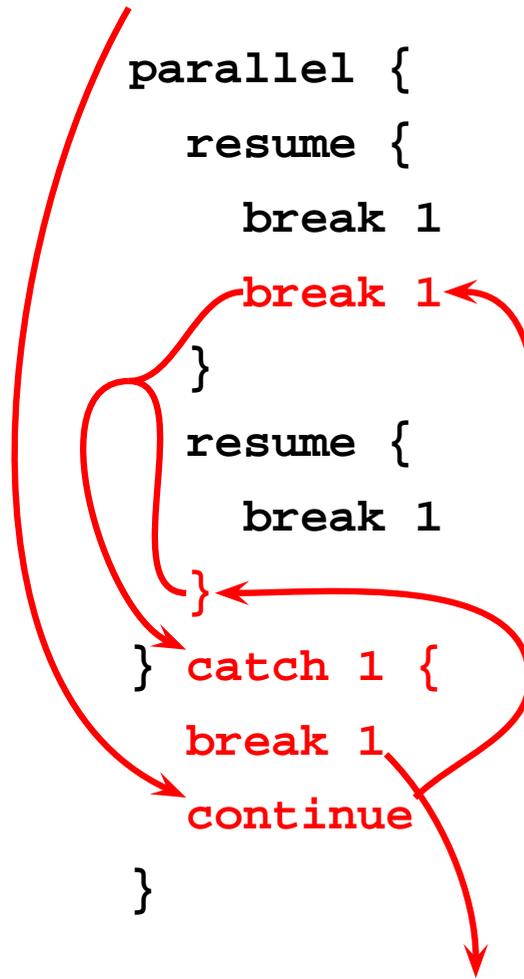
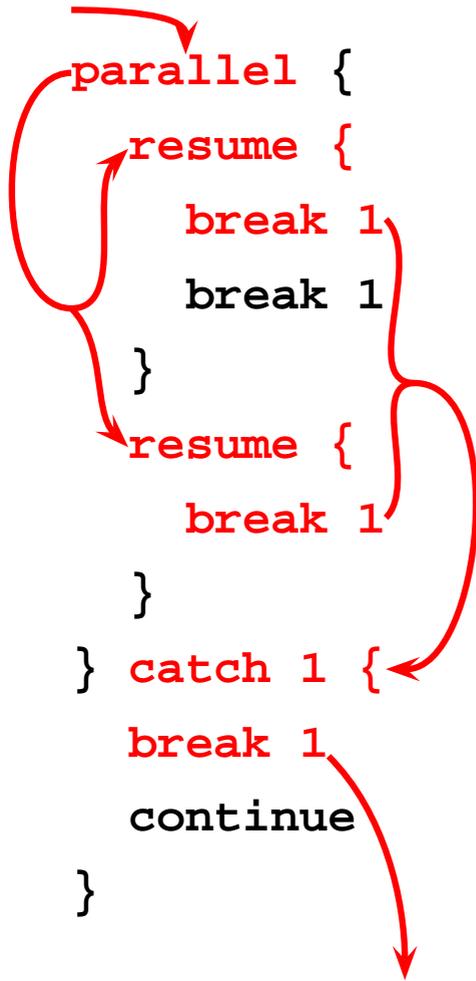
pause

||

pause

```
parallel {  
    resume {  
        break 1  
        break 1  
    }  
    resume {  
        break 1  
    }  
} catch 1 {  
    break 1  
    continue  
}
```

Parallel Behavior



A Minor Point on Completion Codes

Berry's encoding reduces the exit code if it is not handled.

```
try {  
    break 5  
} catch 2 { ... }
```

generates `break 4` in Berry's encoding. I treat it as `break 5`.

I assign each trap its own completion code; they pass unchanged.

Simpler semantics vs. the danger of larger codes.

Irrelevant in HW, probably not a problem for SW.

Conclusions

New ESUIF compiler

- Based on SUIF 2 infrastructure

- Open-source, under development

Intermediate Representation

- Numeric exception codes

- Simple translation into assignments and branches

Future Work on HW & SW Synthesis

- HW/SW synthesis from control dependence
Clever concurrent representation produces efficient hardware and facilitates “sequentializing” SW.
- SW synthesis by static unrolling of cyclic programs
Unrolling SW à la Bourdoncle coupled with constant propagation should quickly execute cyclic programs.
- SW synthesis with dynamic event-based scheduling
Unrolling is expensive if done statically; a scheduler can do it dynamically with little overhead.