Idiom-based Exception Handling using Aspects

Bram Adams
GH-SEL, UGent

Kris De Schutter
PROG, VUB

Outline

1. Idiom-based Exception Handling
2. Analysis
3. Local Continuation Join Point: theory
4. Local Continuation Join Point: practice
5. Manual Recovery
6. Other Aspects
7. Aspicere2
8. Discussion
9. Conclusion

1. Idiom-based Exception Handling (a)

```
int f(int a, int** b){
  int r = OK;
  bool allocated = FALSE;
  r = mem_alloc(10, (int**) b);
  allocated = (r == OK);

  if((r == OK) && ((a < 0) || (a > 10))){
    r = PARAM_ERROR;
    LOG(r,OK); /*root*/
  }
  ...
  if(r == OK){
    r = g(a);
    if(r != OK){
      LOG(LINKED_ERROR,r);
      r = LINKED_ERROR;
    }
  }
  if(r == OK) r = h(b);
  if((r != OK) && allocated)
    mem_free(b);
  return r;
}
```

main logic

rest crosscutting concerns

1. Idiom-based Exception Handling (b)

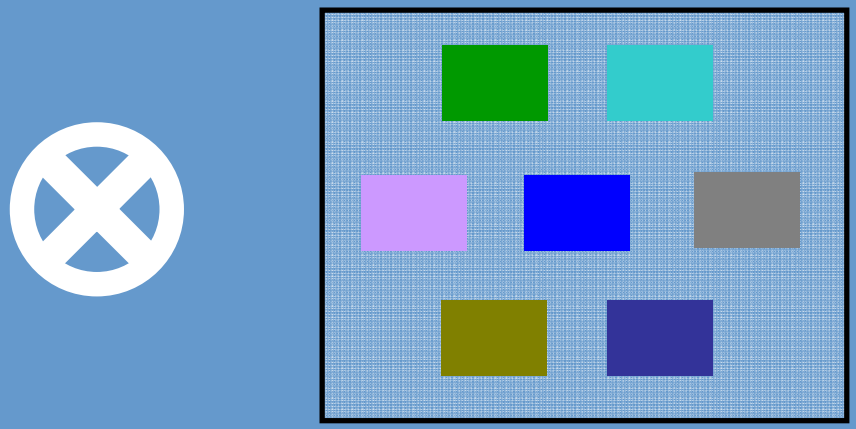
```
/*@range("a",0,10)*/
int f(int a, int** b){
  mem_alloc(10, (int**) b);

  /*@log("LINKED_ERROR")*/
  g(a);
  h(b);
}
```

main logic

bounds checking

logging

aspects

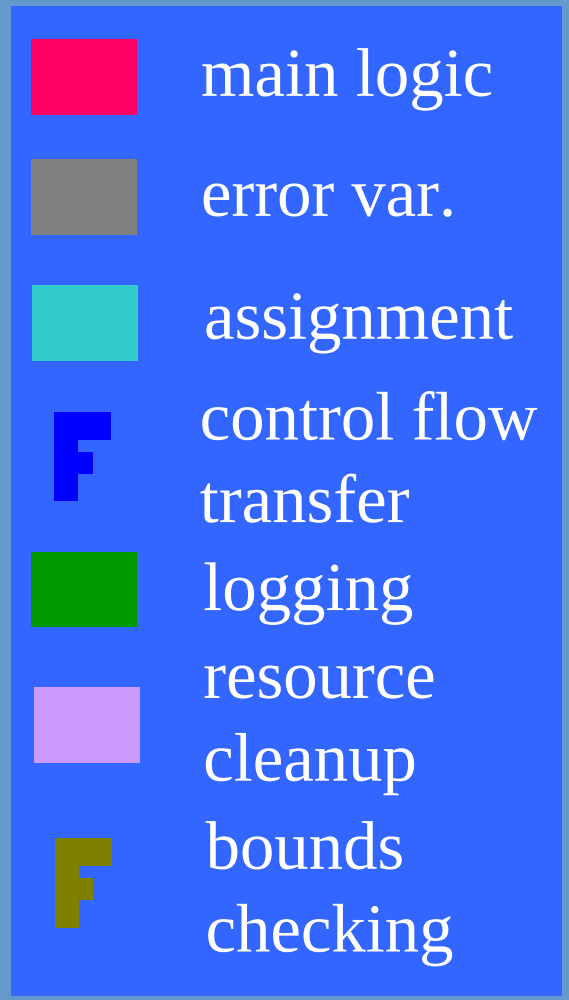


Design decisions:

- aspects are written once
- no aspects written by developers
- annotations configure aspects
- return variables freely available to aspects

2. Analysis (a)

```
int f(int a, int** b){
  int r = OK;
  r = mem_alloc(10, (int**) b);

  if(r != OK){
    /* no logging */
    /* no deallocation */
    return r;
  }else{
    if((a < 0)||(a > 10)){
      r = PARAM_ERROR;
      LOG(r,OK);
      if(r != OK) mem_free(b);
      return r;
    }else{
      r = g(a);
      ...
      if(r != OK){
        LOG(LINKED_ERROR,r);
        r = LINKED_ERROR;
        if(r != OK) mem_free(b);
        return r;
      }else{
        r = h(b);
        if(r != OK){
          /* no logging */
          if(r != OK) mem_free(b);
          return r;
        }else{
          /* no deallocation */
          return r;
        }
      }
    }
  }
}
```



2. Analysis (b)

AOP-alternatives for control flow transfer:

- setjmp/longjmp magic
- continuation passing style
- simple solution:
  - around-advice on each procedure call
  - no proceed() if error happened

} procedure body skipped

```
int f(){
  int i=0;
  do{
    g(&i);
    /*arithmetic and/or I/O on i*/
  }while(i);
  return OK;
}
```

infinite loop

```
int g(int* i_ptr){
  …
  *i_ptr=1;
  …
  return SUDDEN_ERROR;
}
```

3. Local Continuation Join Point: theory (a)

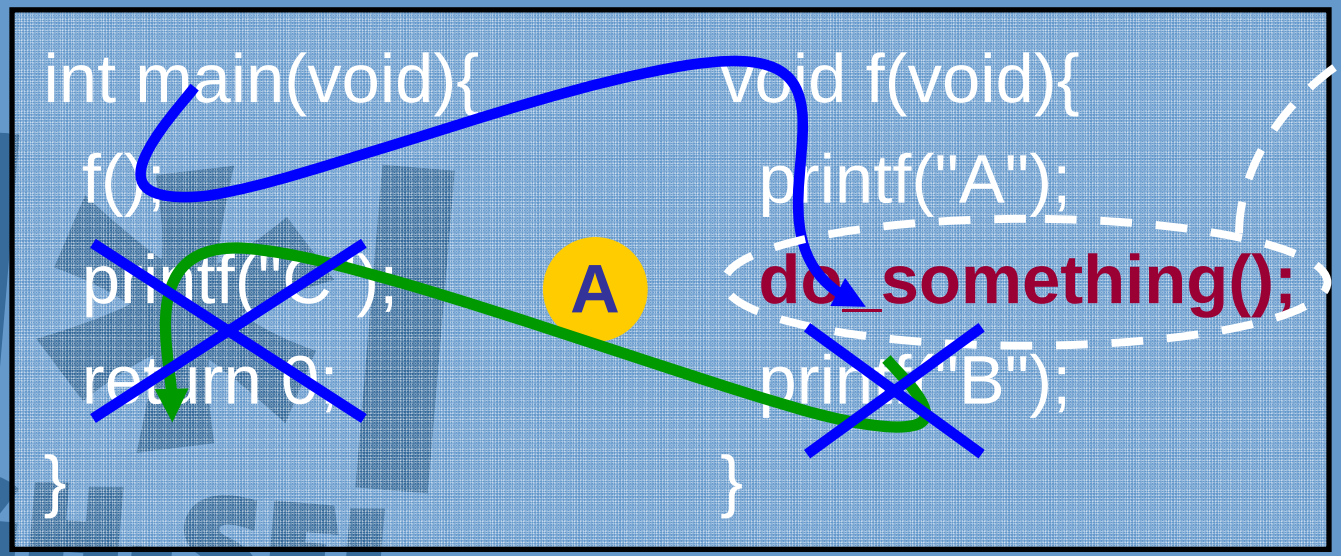
A continuation at any point in the execution of a program P:

the future execution of P from that point on.

target for advice

Continuation of a join point p:

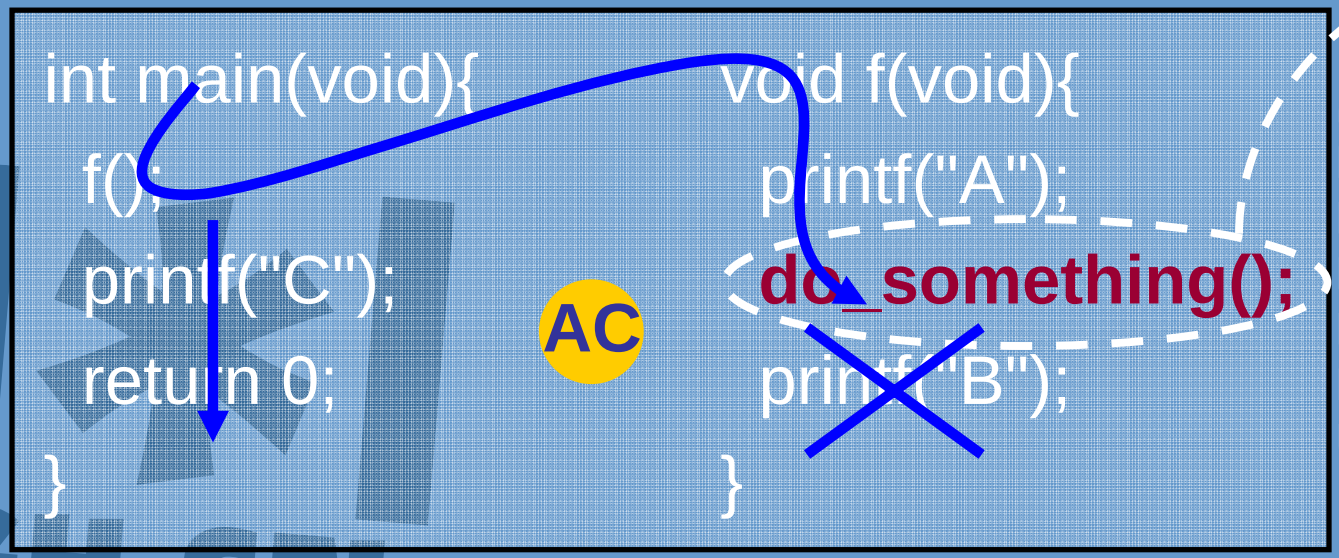
join point representing the future execution after conclusion of p.



3. Local Continuation Join Point: theory (b)

Local continuation of a join point p:

join point representing the future execution after conclusion of p, **limited to the control flow of the procedure** in which p is active.



around LOCAL continuation join point of call{

/*do NOTHING*/

}

4. Local Continuation Join Point: practice (a)

error property

```
int around cflow_transfer(int* R) on Jp:
 idiomatic_call(JpCall,R)
 && !!manual(JpCall)
 && local_continuation(Jp,JpCall){
  if(*R!=OK)
    return *R;
  else
    return proceed();
 }
}
```

skip local continuation

pointcut

advice body

control flow transfer advice (Aspicere2)

4. Local Continuation Join Point: practice (b)

```
int_invocation(Jp,FName):-
  invocation(Jp,FName),
  type(Jp,Type),
  type_name(Type,"int")
  .

idiomatic_proc(Jp):-
  execution(Jp,_),
  filename(Jp,"main.c")
  .
```

Prolog predicates

```
idiomatic_call(Jp,R):-
  int_invocation(Jp,FName),
  \+wildcard(".*printf",FName),
  enclosingMethod(Jp,JpEncl),
  idiomatic_proc(JpEncl),
  property(JpEncl,error_var,R)
  .
```

limit scope of aspects to idiomatic modules

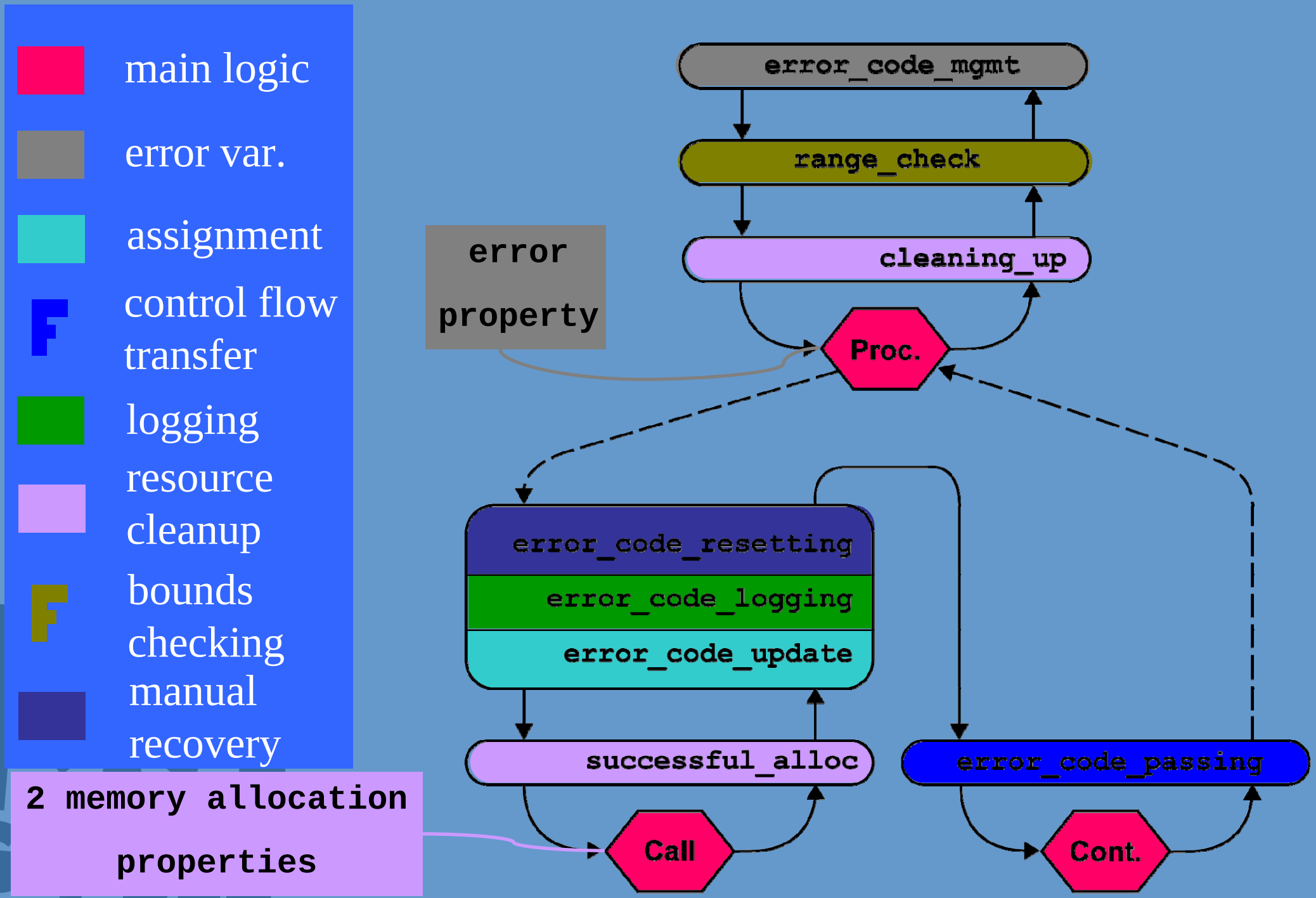
exclude (standard) libraries

5. Manual Recovery

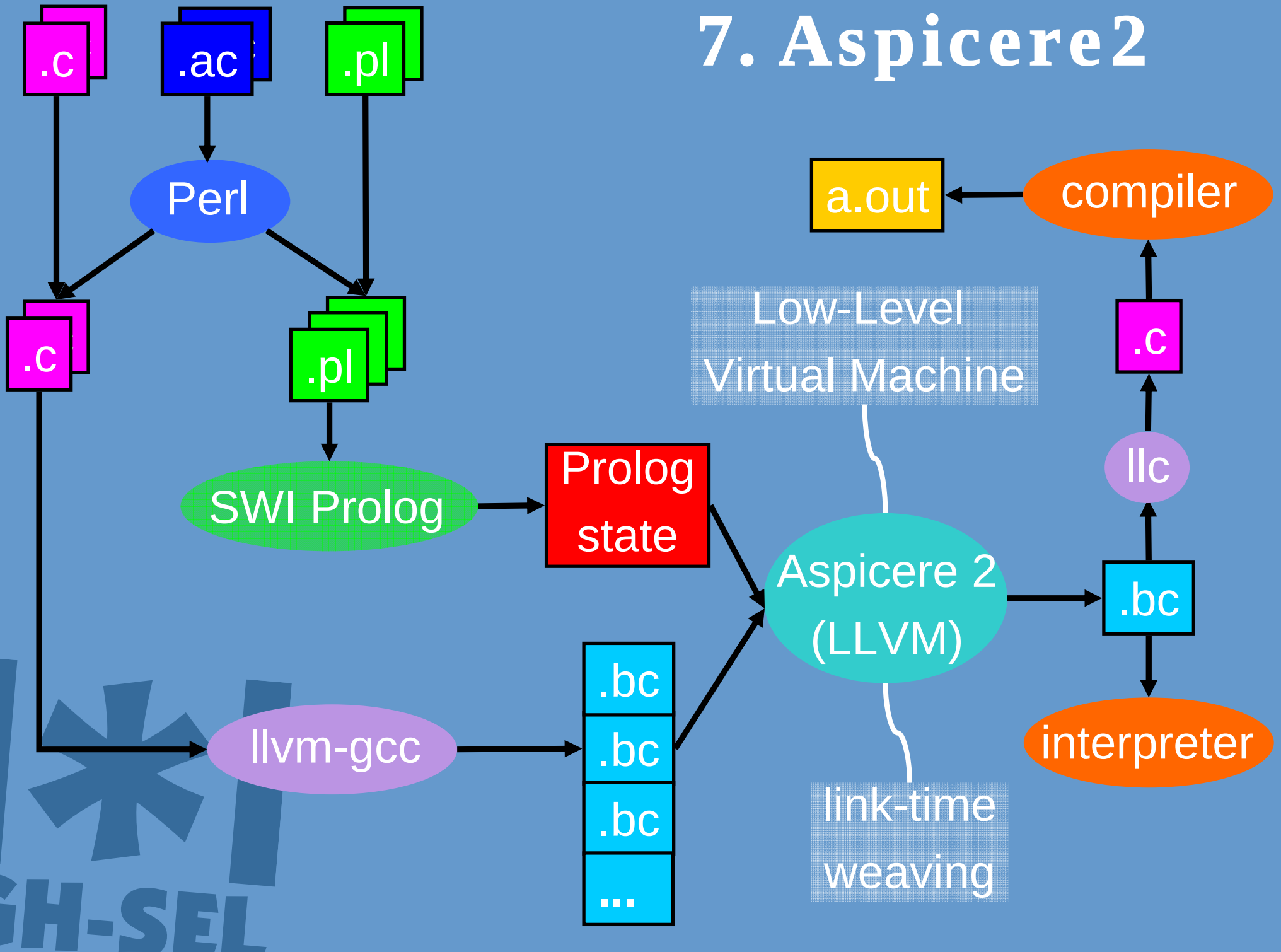
```
int f(void){
  int error=OK;

  /*@manual()*/
  error=g();

  if(error==EASY_TO_FIX){
    /* full manual recovery */
  }else if(error==EXTRA_CLEANUP){
    /* do some initial recovery */
    rethrow(error);
  }
  ...
}
```

```
int rethrow(int a){
  return a;
}
```

6. Other Aspects



7. Aspicere2


.c
.ac
.pl
Perl
.c
.pl
SWI Prolog
Prolog state
llvm-gcc
.bc
.bc
.bc
...
Low-Level Virtual Machine
Aspicere 2 (LLVM)
link-time weaving
a.out
compiler
.c
llc
.bc
interpreter

8. Discussion (a)

Code size estimation:

- 20 kLOC module of which 1716 LOC of exception handling [1]
- aspects together with Prolog files account for 122 LOC
- @log-annotation for each logged linked error
- @manual-annotation + manual recovery code

Migration (cf. [1]):

- find actual main concern and the relevant error values
- remove error handling code
- insert annotations

Generic exception handling advice:

- use of context variables for types, annotation attributes, etc.
- **robust pointcuts** based on:
  - returning an integer;
  - local continuation join points;
  - annotations;
  - join point properties.

[1] M. Bruntink, A. van Deursen, and T. Tourwé. An initial experiment in reverse engineering aspects. In Proc. of WCRE '04, pages 306-307. IEEE Computer Society, 2004.

8. Discussion (b)

Costs of our approach:

- **build-time overhead (± factor 10)**
- **run-time overhead (± 10%)**:
  - advice is transformed into procedures;
  - cleanup aspect adds extra local variables.

Benefits of our AOP-solution:

- switch aspects to change exception handling strategy
- code readability and evolvability
- optimisation:
  - join point properties can be mapped onto local variables;
  - advice on local continuation join points can be inlined efficiently;
  - bounds checking aspect faster than idiom;
  - bytecode optimisation passes.

9. Conclusion

Aspects:

- hide return-code idiom administration ...
- ... unless developer wants to do manual recovery

Benefits:

- centered around local continuation join points
- fairly robust pointcuts and advice
- improved code understandability and evolvability

Costs:

- limited run-time and build penalty