DEBUGGING: STRUCTURED DEBUGGING

WS 2017/2018

Martina Seidl
Institute for Formal Models and Verification

JOHANNES KEPLER
UNIVERSITY LINZ

Cause of a Failure

The cause of any event (“effect”) is a preceding event without which the effect would not have occurred.

to prove causality, one must show that

- the effect occurs when the cause occurs
- the effect does not occur when the cause does not.

advantages in programming

- programs are (high-level) abstractions of reality
- program runs are usually repeatable
- testing can be automated

Debugging: Ad-Hoc Approach

guess the cause of a failure based on

- intuition
- experience

problems with this approach

- a priori knowledge is necessary
- hardly systematic
- hardly reproducible
- hard to teach

challenge: systematically find an explanation for a failure

Debugging: Scientific Method

process of obtaining a theory that explains some aspects of the universe

process outline:

1. observe a failure
2. establish a hypothesis that is consistent with observations
3. make predictions based on the hypothesis
4. test the hypothesis by experiments and further observations
   - refine hypothesis if experiment satisfy the predictions
   - otherwise, create alternative hypothesis
5. repeat 3. and 4. until no refinement is possible

Example: Broken Shell Short

```
int main (int argc, char *argv []) {
    int *a;
    int i;

    a = (int *)malloc((argc - 1) * sizeof(int));
    for (i = 0; i < argc - 1; i++)
      a[i] = atoi(argv[i + 1]);

    shell_sort(a, argc);

    printf("Output: ");
    for (i = 0; i < argc - 1; i++)
      printf("%d ", a[i]);
    printf("\n");

    free(a);
    return 0;
}
```

Example: Broken Shell Short

```
int main (int argc, char *argv []) {
    int *a;
    int i;

    a = (int *)malloc((argc - 1) * sizeof(int));
    for (i = 0; i < argc - 1; i++)
      a[i] = atoi(argv[i + 1]);

    shell_sort(a, argc);

    printf("Output: ");
    for (i = 0; i < argc - 1;
      printf("%d ", a[i]);
    printf("\n");

    free(a);
    return 0;
}
```

Preparation:

hypothesis	input “11 14” works
prediction	output is “11 14”
experiment	run with input “11 14”
observation	output is “0 11”
conclusion	hypothesis rejected

Example: Broken Shell Short

```
int main (int argc, char *argv []) {
 1    int *a;
 2    int i;

 3    a = (int *)malloc((argc - 1) * sizeof(int));
 4    for (i = 0; i < argc - 1; i++)
        a[i] = atoi(argv[i + 1]);

 5    shell_sort(a, argc);

 6    printf("Output: ");
 7    for (i = 0; i < argc - 1;
        printf("%d ", a[i]);
 8    printf("\n");

 9    free(a);
10    return 0;
}
```

Hypothesis 1:

hypothesis	`a[0]` becomes zero
prediction	`a[0] = 0` in line 9
experiment	observe `a[0]`
observation	`a[0] = 0`
conclusion	hypothesis confirmed

Example: Broken Shell Short

```
int main (int argc, char *argv []) {
 1    int *a;
 2    int i;

 3    a = (int *)malloc((argc - 1) * sizeof(int));
 4    for (i = 0; i < argc - 1;
        a[i] = atoi(argv[i + 1]

 5    shell_sort(a, argc);

 6    printf("Output: ");
 7    for (i = 0; i < argc - 1;
        printf("%d ", a[i]);
 8    printf("\n");

 9    free(a);
10    return 0;
}
```

Hypothesis 2:

hypothesis	infection in `shell_sort`
prediction	`a = [11, 14]`, `size = 2` in line 5
experiment	observe `a, size`
observation	`a = [11, 14, 0]`, `size = 3`
conclusion	hypothesis rejected

Example: Broken Shell Short

```
int main (int argc, char *argv []) {
    int *a;
    int i;

    a = (int *)malloc((argc - 1) * sizeof(int));
    for (i = 0; i < argc - 1;
      a[i] = atoi(argv[i + 1]

    shell_sort(a, argc);

    printf("Output: ");
    for (i = 0; i < argc - 1;
      printf("%d ", a[i]);
    printf("\n");

    free(a);
    return 0;
}
```

Hypothesis 3:

hypothesis	`size = 3` causes failure in `shell_sort`
prediction	if we set `size = 2` program works
experiment	set `size = 2`
observation	as predicted
conclusion	hypothesis confirmed

Example: Broken Shell Short

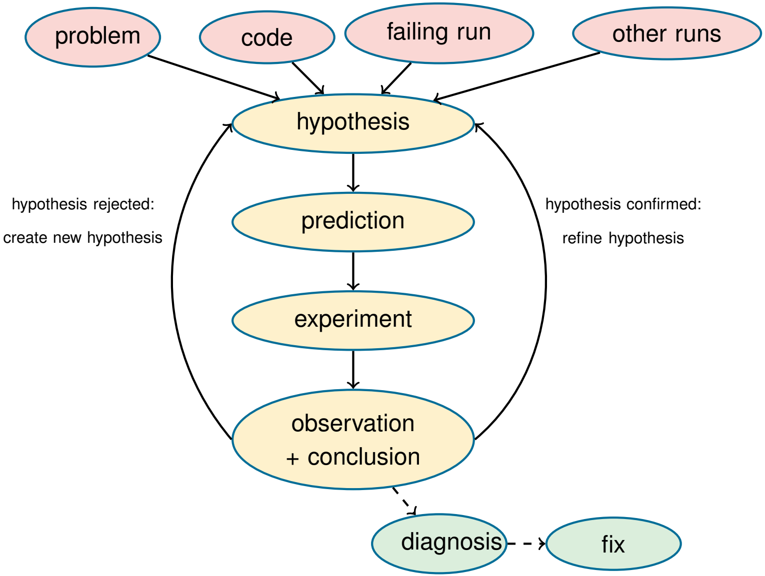
```
int main (int argc, char *argv []) {
 1     int *a;
 2     int i;

 3     a = (int                    - 1) * sizeof(int));
 4     for (i = 0; i < argc - 1; i++)
         a[i] = atoi(argv[i + 1]

 5     shell_sort(a, argc);

 6     printf("Output: ");
 7     for (i = 0; i < argc - 1;
         printf("%d ", a[i]);
 8     printf("\n");

 9     free(a);
10     return 0;
}
```

change to `argc-1`

Hypothesis 4:

hypothesis	using `argc` instead of `argc-1` in `shell_sort` causes failure
prediction	output is “11 14”
experiment	change `argc` to `argc-1` in line 5
observation	as predicted
conclusion	hypothesis confirmed

Summary: Scientific Method



Deriving a Hypothesis

- **problem description**: without concise description, the problem cannot be solved
- **program code**: common abstraction across all program runs
- **failing run**: execute the code and reproduce the problem observe actual facts about the concrete run
- **alternate runs**: identification of anomalies — differences between failing run and passing runs
- **earlier hypotheses**:
  - include passed hypotheses
  - exclude failed hypotheses

Theories in Debugging

When the hypothesis explains all experiments and observations, the hypothesis becomes a theory.

a theory is a hypothesis that

- explains earlier observations
- predicts further observations

context of debugging: a theory is called a diagnosis

This contrasts popular usage, where a theory is a vague guess

Algorithmic Debugging

basic idea: (partially) automate the debugging process by interactively querying the user about infection sources

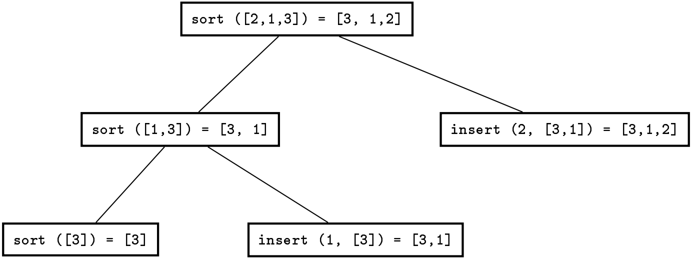
approach:

1. assume an incorrect result R with origins $O_1, O_2, \ldots, O_n$
2. for each O_i, enquire whether O_i is correct
3. if some O_i is incorrect, continue at Step 1 with $R = O_i$
4. otherwise (all O_i are correct), we found the defect

Example: Algorithmic Debugging

```
def insert (elem, list):
    if len (list) == 0:
        return [elem]
    head = list[0]
    tail = list[1:]
    if elem <= head:
        return list + [elem]
    return [head] + insert (elem, tail)
```

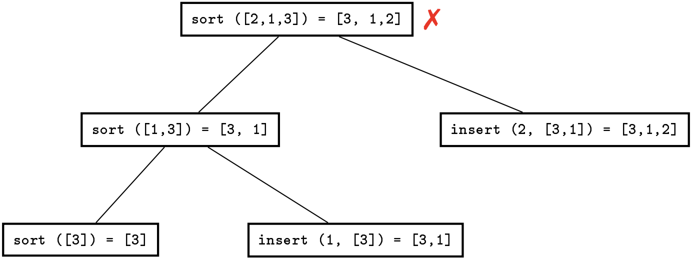
```
def sort (list):
    if len (list) <= 1:
        return list
    head = list[0]
    tail = list[1:]
    return insert (head, sort(tail))
```



Example: Algorithmic Debugging

```
def insert (elem, list):
    if len (list) == 0:
        return [elem]
    head = list[0]
    tail = list[1:]
    if elem <= head:
        return list + [elem]
    return [head] + insert (elem, tail)
```

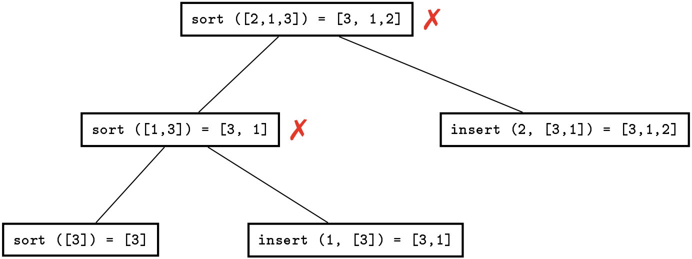
```
def sort (list):
    if len (list) <= 1:
        return list
    head = list[0]
    tail = list[1:]
    return insert (head, sort(tail))
```



Example: Algorithmic Debugging

```
def insert (elem, list):
    if len (list) == 0:
        return [elem]
    head = list[0]
    tail = list[1:]
    if elem <= head:
        return list + [elem]
    return [head] + insert (elem, tail)
```

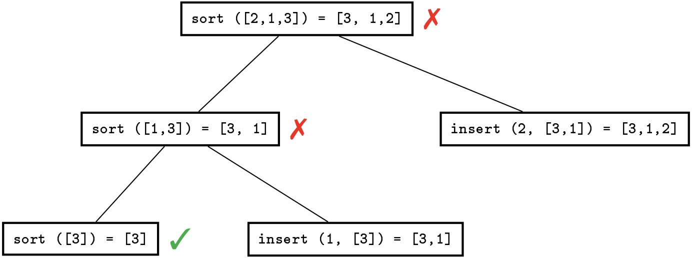
```
def sort (list):
    if len (list) <= 1:
        return list
    head = list[0]
    tail = list[1:]
    return insert (head, sort(tail))
```



Example: Algorithmic Debugging

```
def insert (elem, list):
    if len (list) == 0:
        return [elem]
    head = list[0]
    tail = list[1:]
    if elem <= head:
        return list + [elem]
    return [head] + insert (elem, tail)
```

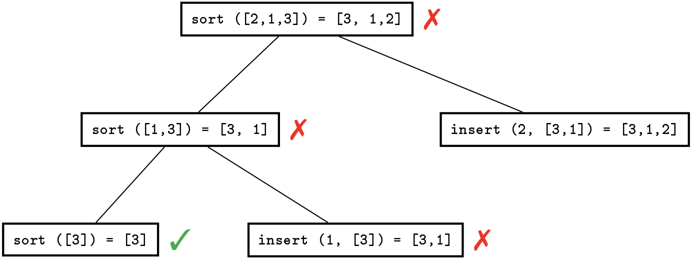
```
def sort (list):
    if len (list) <= 1:
        return list
    head = list[0]
    tail = list[1:]
    return insert (head, sort(tail))
```



Example: Algorithmic Debugging

```
def insert (elem, list):
    if len (list) == 0:
        return [elem]
    head = list[0]
    tail = list[1:]
    if elem <= head:
        return list + [elem]
    return [head] + insert (elem, tail)
```

```
def sort (list):
    if len (list) <= 1:
        return list
    head = list[0]
    tail = list[1:]
    return insert (head, sort(tail))
```



Algorithmic Debugging: Critical Discussion

- drive the search for a deffect in a systematic way guided by human input
- problems on real-world scenarios:
  - scalability: number of functions, shared data structures, ...
    $\Rightarrow$ works best for functional and logical programming languages
  - process is too mechanical: programmer has to assist the tool

$\Rightarrow$ replace programmer by oracle that knows the external specification of the program

Structuring the Debugging Process

not every problem needs the strength of a the scientific method, but for complex problems it is useful to

- be explicit is important to understand (and find) the problem
- write down hypotheses and observations in order to know
  - where you are
  - where you have been
  - where you want to go
  - what you want to get

Reasoning about Programs for Debugging

- **deduction** (0 runs)
  reason from (abstract) program code to concrete runs
  $\Rightarrow$ static analysis
- **observation** (1 run)
  inspection of a single program run
  $\Rightarrow$ facts about program execution
- **induction** (n runs)
  reasoning from the particular to the general
  $\Rightarrow$ summary of findings from multiple runs
- **experimentation** (n controlled runs)
  refinement and rejection of hypotheses
  $\Rightarrow$ scientific method