COMP0104 Software Development Practice: Static Analysis

Jens Krinke

Centre for Research on Evolution, Search & Testing
Software Systems Engineering Group
Department of Computer Science
University College London

Static Analysis Tools

- Static analysis tools support the development and maintenance of source code.
- These tools work **statically**, they examine program code without executing it.

Fibonacci

The **Fibonacci** function *fib*(n) calculates the nth member of the Fibonacci sequence
1, 1, 2, 3, 5, 8, ...,
in which every element is defined as the sum of its two ancestors.

It is recursively defined as:

$$fib(n) = \begin{cases} 1, & \text{for } n = 0 \vee n = 1 \\ fib(n-1) + fib(n-2), & \text{otherwise} \end{cases} .$$

```c
#include <stdio.h>
int fib (int n)
{
	int f, f0 = 1, f1 = 1;
	while (n > 1) {
		n = n - 1;
		f = f0 + f1;
		f0 = f1;
		f1 = f;
	}
	return f;
}

int main ()
{
	int n = 10;
	while (n-- > 1)
		(void) printf("fib(%d)=%d\n", n, fib(n));
	return 0;
}
```

What happens here?

```
$ cc -o fibo fibo.c
$ ./fibo
fib(9)=55
fib(8)=34
fib(7)=21
fib(6)=13
fib(5)=8
fib(4)=5
fib(3)=3
fib(2)=2
fib(1)=134513905
$ _
```

The compiler is quiet

```
$ gcc -o fibo fibo.c -Wall
$ _
```

However, with optimization (-O):

```
$ gcc -o fibo fibo.c -Wall -O
fibo.c: In function ‘fibonacci’:
fibo.c:3: warning: ‘f’ might be used uninitialized in
this function
$ _
```

Data flow analysis

Data flow analyses collect information about the use of values and variables of a program that is as accurate as possible.

The data flow information produced

- is needed mainly in program **optimisation**, to identify which variables are best kept in registers and when the values of the variables are actually needed.
- can also be used to detect uninitialised variables and other programming mistakes.

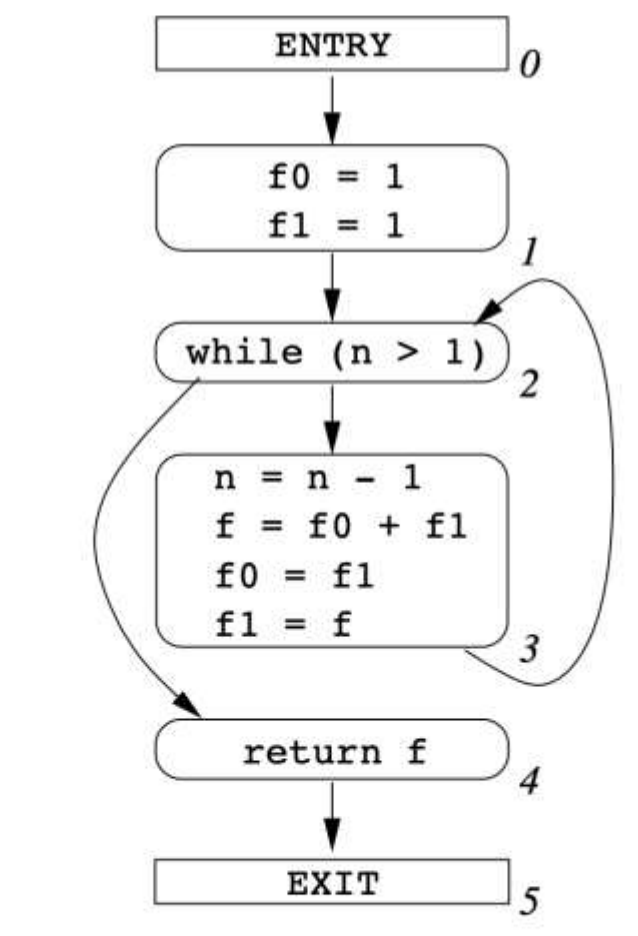
Control flow graph

A control flow graph shows in which order the statements can be executed:

- The statements are mapped to nodes.
- A series of statements that must be executed in sequence can be combined into a **basic block**.
- Edges connecting the nodes represent the possible control flow between the instructions.
- An entry and exit node represent the beginning and the end of the program or function.

Example

```c
#include <stdio.h>
int fib (int n)
{
	int f, f0 = 1, f1 = 1;
	while (n > 1) {
		n = n - 1;
		f = f0 + f1;
		f0 = f1;
		f1 = f;
	}
	return f;
}
```



Data flow analysis via data flow sets

- *gen*(*S*) is the **generated** set:
  the set of variables newly defined in node *S*.
- *kill*(*S*) is the **destroyed** set:
  the set of variables that are made undefined in a node, their status is unknown after the node.
- *in*(*S*) is the **incoming** set, the set of variables that are defined before entering the node.
- *out*(*S*) is the **outgoing** set, the set of variables that are defined at the end of the node.

Data flow equations describe the relation between the sets

- Outgoing data flow *out*(*S*):

$$out(S) = gen(S) \cup (in(S) - kill(S))$$

- Incoming data flow *in*(*S*):

$$in(S) = \bigcap_{S_i \in \mathbf{pred}(S)} out(S_i)$$

the data flow that **must** be present

Example:
initialised variables in the fibonacci function

- We will start with the determination of *gen*(S) and *kill*(S) for every node S.
- *in*(S) and *out*(S) cannot be immediately computed because of a cycle in the control flow graph; they are recursively dependent on each other.

Example:
initialised variables in the fibonacci function

We solve this problem using **fix point iteration**:

- starting from suitable initial values
- alternately determine *in*(*S*) / *out*(*S*) in several runs,
- until the values no longer change.

What is a suitable initialisation?

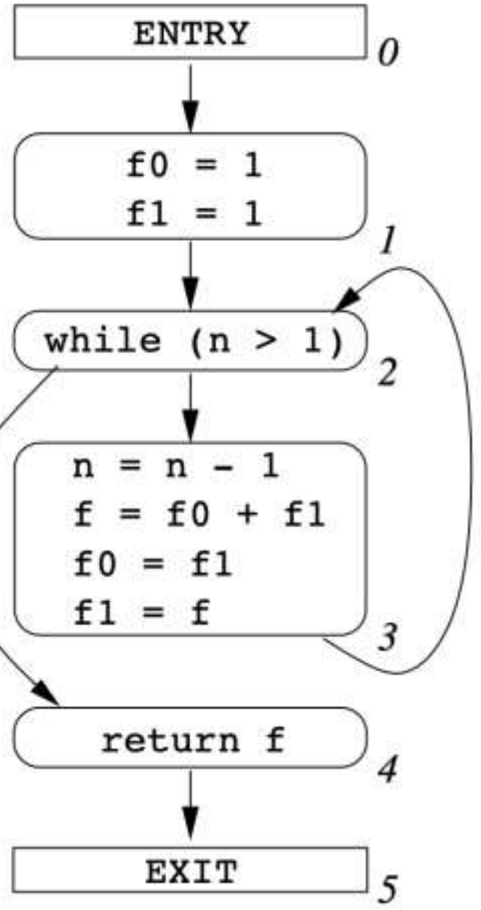
$$in(S) = \bigcap_{S_i \in \mathbf{pred}(S)} out(S_i)$$

Assume two predecessors: S_1 and S_2

- S_1 is known, S_2 is "unknown"
- It should hold: $S_1 \cap S_2 = S_1$
- We use a new "top" element: $\top$
- For all d: $d \cap \top = d$

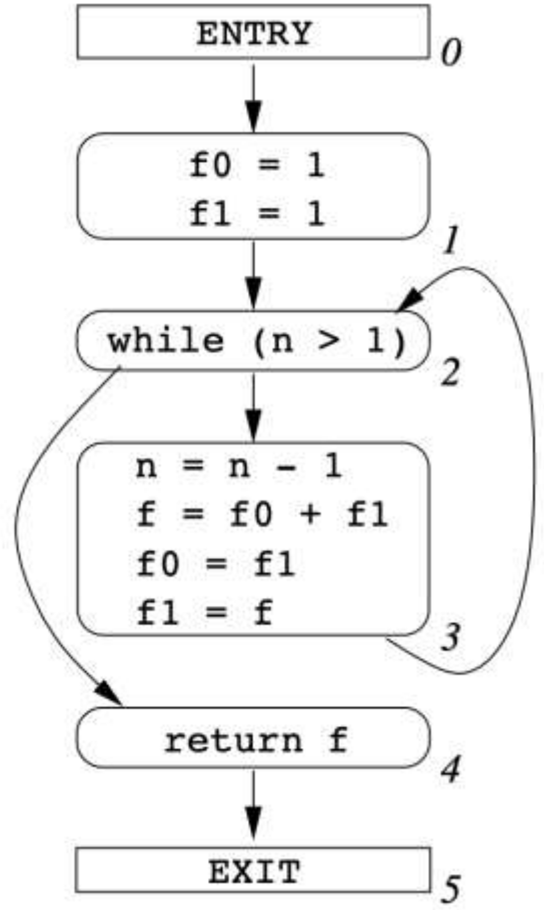
Example

	gen(S)
S_0	{n}
S_1	{f0,f1}
S_2	{}
S_3	{n,f,f0,f1}
S_4	{}
S_5	{}



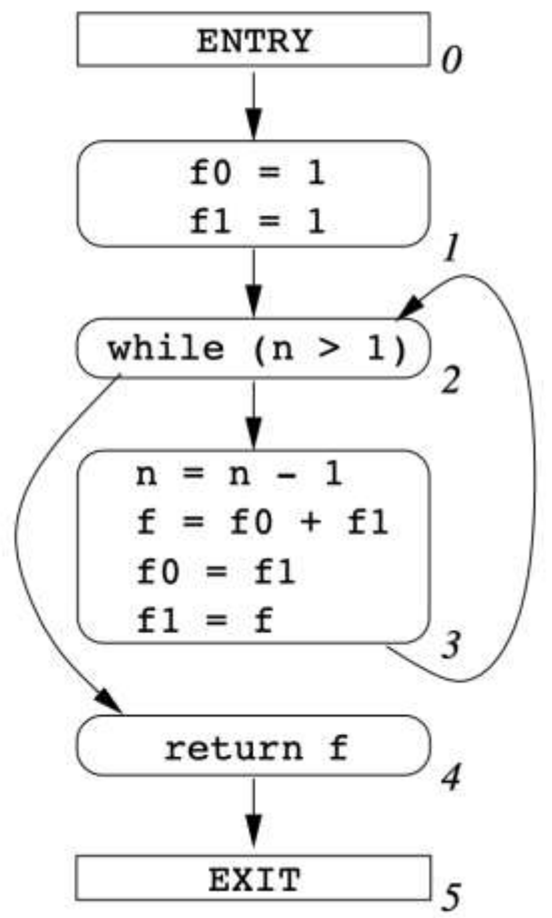
Example

	gen(S)	*in(S)*	*out(S)*
S_0	{n}	T	T
S_1	{f0,f1}	T	T
S_2	{}	T	T
S_3	{n,f,f0,f1}	T	T
S_4	{}	T	T
S_5	{}	T	T



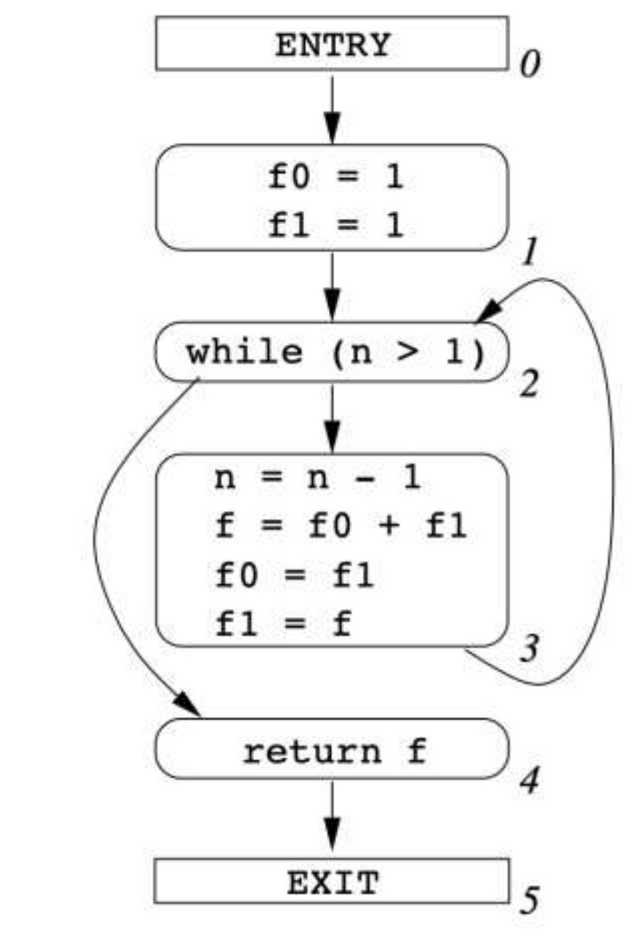
Example: 1st run

	gen(*S*)	*in*(*S*)	*out*(*S*)
S_0	{n}	{}	{n}
S_1	{f0,f1}	{n}	{n,f0,f1}
S_2	{}	{n,f0,f1}	{n,f0,f1}
S_3	{n,f,f0,f1}	{n,f0,f1}	{n,f,f0,f1}
S_4	{}	{n,f0,f1}	{n,f0,f1}
S_5	{}	{n,f0,f1}	{n,f0,f1}



Example: 2nd run

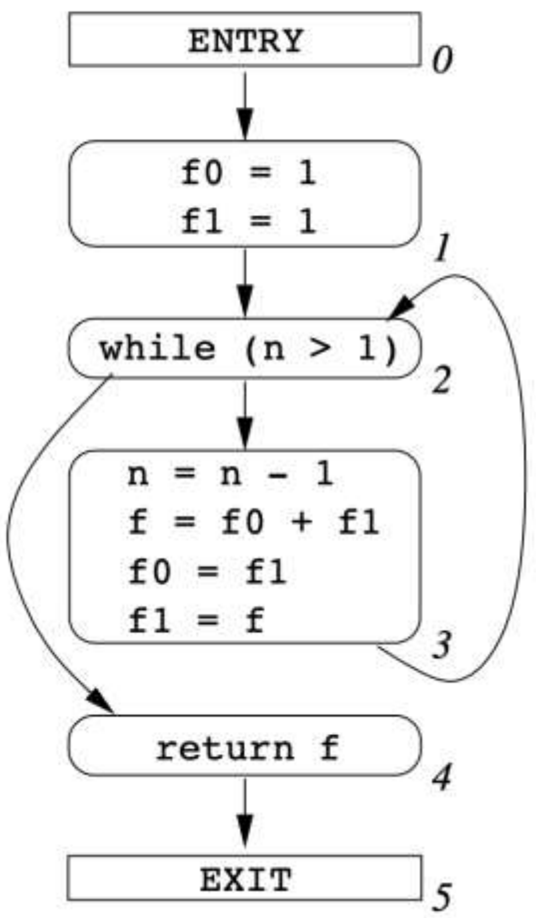
	gen(S)	*in*(S)	*out*(S)
S_0	{n}	{}	{n}
S_1	{f0,f1}	{n}	{n,f0,f1}
S_2	{}	{n,f0,f1}	{n,f0,f1}
S_3	{n,f,f0,f1}	{n,f0,f1}	{n,f,f0,f1}
S_4	{}	{n,f0,f1}	{n,f0,f1}
S_5	{}	{n,f0,f1}	{n,f0,f1}



And what do the final values tell us?

The set $in(S_4)$ does not contain the variable f, although `f` is a return value.

➡It is therefore possible that the returned value is undefined.



Enable warnings for your compiler!

- **All** warning possibilities of the compiler should **always** be used.
- Every warning must be investigated, and the code improved until the last warning has disappeared.
- There is a danger that new warnings are overlooked
  - if warnings are left,
  - or if the warnings are switched off completely.

Bugs are usually “small”

- Competent Programmer Hypothesis:
  Programs are very close to a correct version, or that the difference between current and correct code for each fault is very small.
- Programmers are not stupid,
  but they make stupid mistakes…
- Most stupid mistakes can be easily identified!
- `if (p == null && p.continue()) …`
- `if (str == “ok”) …`
- `if (p == null | p.stop()) …`

FindBugs / SpotBugs

- FindBugs analyses the bytecode of Java classes and detects potential bugs.
- FindBugs implements a large range of detectors which target specific issues.
- The detectors use a variety of inspection techniques, from checking the structure of the class to full dataflow analysis.
- It is possible to extend FindBugs with new detectors.
- SpotBugs checks for more than 400 bug patterns.

FindBugs

- Initially developed at the University of Maryland
- Is in use by Google, eBay, …
- eBay found that 2 developers reviewing FindBugs was 10 times more effective than 2 testers.

Bug Descriptions: Bad Practice

- Violations of recommended and essential coding practice.
- Example:
  Comparison of String parameter using == or !=

Bug Descriptions: Correctness

- Probable bug – an apparent coding mistake resulting in code that was probably not what the developer intended.
- Example:
  An apparent infinite loop.

Bug Descriptions: Internationalization

- Code flaws having to do with internationalization and locale.
- Example:
  Reliance on default encoding.

Bug Descriptions:
Malicious code vulnerability

- Code that is vulnerable to attacks from untrusted code.
- Example:
  Field isn’t final but should be.

Bug Descriptions:
Multithreaded correctness

- Code flaws having to do with threads, locks, and volatiles.
- Example:
  Static DateFormat

Bug Descriptions: Performance

- Code that is not necessarily incorrect but may be inefficient.
- Example:
  Private method is never called.

Bug Descriptions: Security

- A use of untrusted input in a way that could create a remotely exploitable security vulnerability.
- Example:
  A prepared SQL statement is generated
  from a nonconstant String.

Bug Descriptions: Dodgy Style

- Code that is confusing, anomalous, or written in a way that leads itself to errors.
- Example:
  Class doesn’t override equals in superclass

A good source for dodgy code

The Daily WTF recounts tales of disastrous development, from project management gone spectacularly bad to inexplicable coding choices.

```c
/* Fibonacci */
int fib (int n)
{
	int f, f0 = 1, f1 = 1;
	while (n > 1) {
		n = n - 1;
		f = f0 + f1;
		f0 = f1;
		f1 = f;
	}
	return f;
}
```

What was the programmer’s intention?

There are two possible reasons for the problem.

- the programmer could have forgotten to initialise `f`, because if `f` is initialised with the value 1, the Fibonacci function returns the correct values.
- it may also be that the function is only used for values larger than one, since the values for 0 and 1 are defined explicitly.

→The programmer should have dealt with this kind of exception by using **assertions**.

```c
#include <assert.h>
/* Fibonacci */
int fib (n)
{
      int f, f0 = 1, f1 = 1;
      assert(n > 1);
      while (n > 1) {
            n = n - 1;
            f = f0 + f1;
            f0 = f1;
            f1 = f;
      }
      return f;
}
```

```
$ cc -o fibo-assert fibo-assert.c
$ ./fibo-assert
fib(9)=55
fib(8)=34
fib(7)=21
fib(6)=13
fib(5)=8
fib(4)=5
fib(3)=3
fib(2)=2
fibo-assert.c:6: failed assertion `n > 1'
sh: 2040 IOT instruction
     (core dumped)  fibo-assert
$ _
```

Disadvantages of using assertions as specifications

- Assertions are checked **dynamically**, i.e. faulty behaviour can only be recognised by executing test cases.
- Assertions are limited to properties that can be expressed as a C expression.
- Assertions are in the code, not in the interface documentation (the header file).

Runtime Exceptions

- Throwing a runtime exception is often a reasonable way to fail safely and report a failure.
- Runtime exceptions represent conditions that reflect errors in your program's logic and cannot be reasonable recovered from.
- Examples: NullPointerException, IllegalStateException, IllegalArgumentException…
- Mistakes that fail silently are expensive mistakes!

Concepts

- **Static program analysis** can identify common errors at compile time.
- As a side effect of the program optimisation, **data flow analysis** can reveal the use of uninitialised variables.
- **Assertions** guard against incorrect program status. They can be used to check the compliance of pre and post conditions during runtime.
- Throwing **a runtime exception** is often a reasonable way to fail safely and report a failure.