Deductive Verification of Data Structures

Jens Gerlach

DEVICE-SOFT Workshop

Berlin, 21./22. October 2010

Introduction

- I will mostly talk about a particular and well known data type

`stack`

- The principles to be discussed can be applied to other data structures as well

Fraunhofer FIRST

Overview

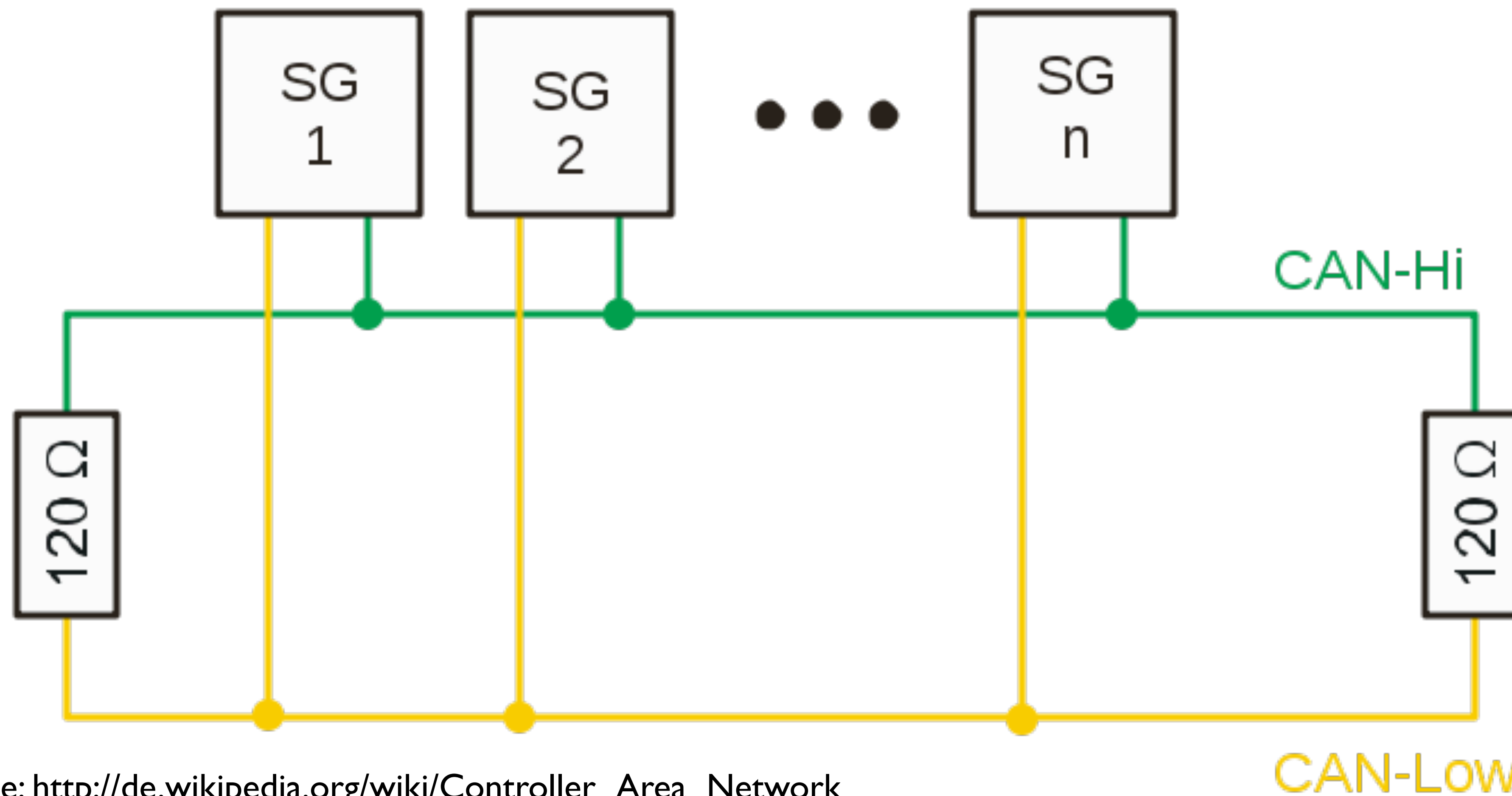
- Motivation
- A simple C stack
- From unit tests to unit proofs
- Representation independence
- Conclusions

Fraunhofer FIRST

Motivation

Fraunhofer
FIRST

Vehicle Networks

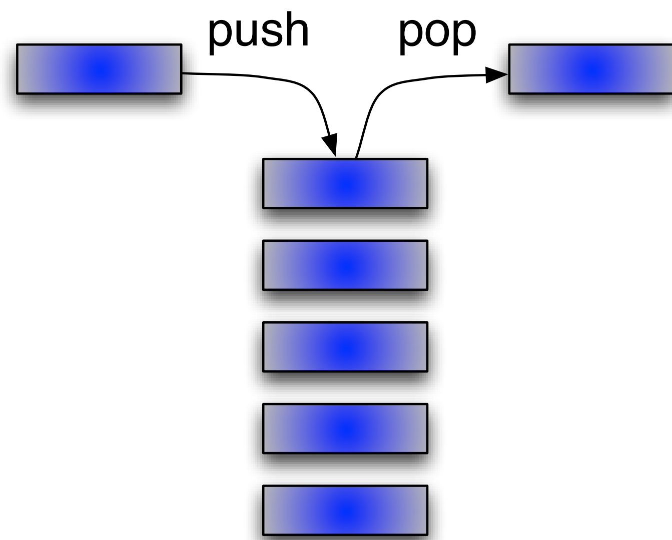


Source: http://de.wikipedia.org/wiki/Controller_Area_Network

Fraunhofer FIRST

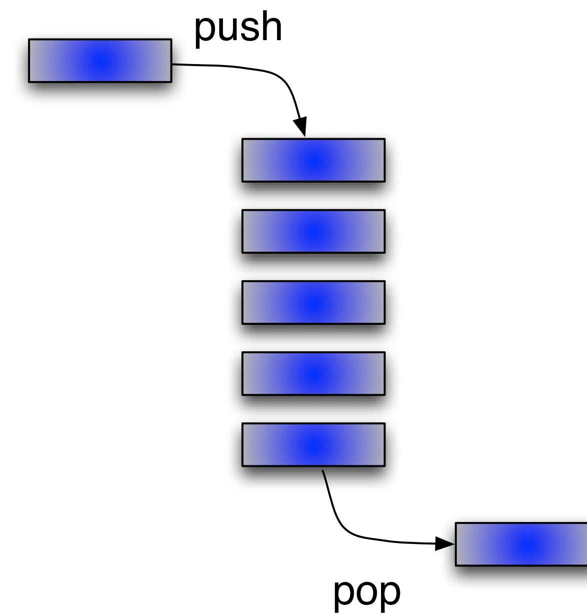
Buffering of Messages

Stack



first in, last out

Queue



first in, first out

Fraunhofer
FIRST

C++ standard library

- `std::stack<T>` and `std::queue<T>`
  - generic and type safe
  - portable and operationally proven
- disadvantages (for embeddes systems)
  - not C, dynamic memory

Fraunhofer
FIRST

A simple C Stack

Fraunhofer
FIRST

Design guide lines

- interface similar to std::stack
- can only hold a maximum number of elements
  - no need for dynamic memory

Fraunhofer
FIRST

What exactly is a stack?

- a data type with associated functions
  - `create`
  - `empty`, `full`
  - `push`, `top`, `pop`,
- functions must satisfy *stack axioms*

Fraunhofer
FIRST

Some stack axioms

```
IsEmpty(create()) = TRUE

IsEmpty(push(x,s)) = FALSE

pop(push(x,s)) = s

top(push(x,s)) = x

push(top(s),pop(s)) = s, if s not empty
```

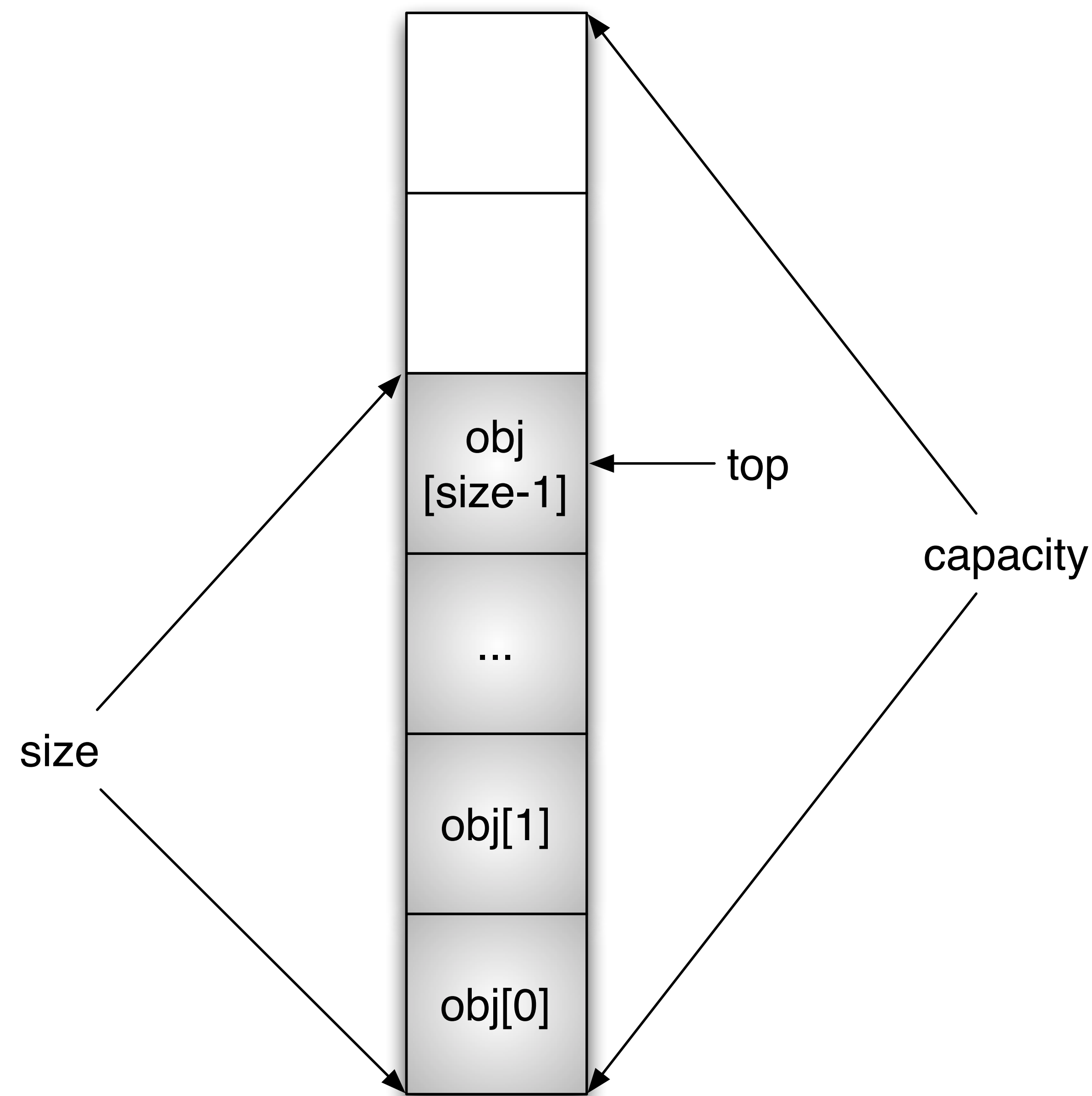
Fraunhofer
FIRST

Focus on Axiom 3

```
pop(push(x,s)) = s
```

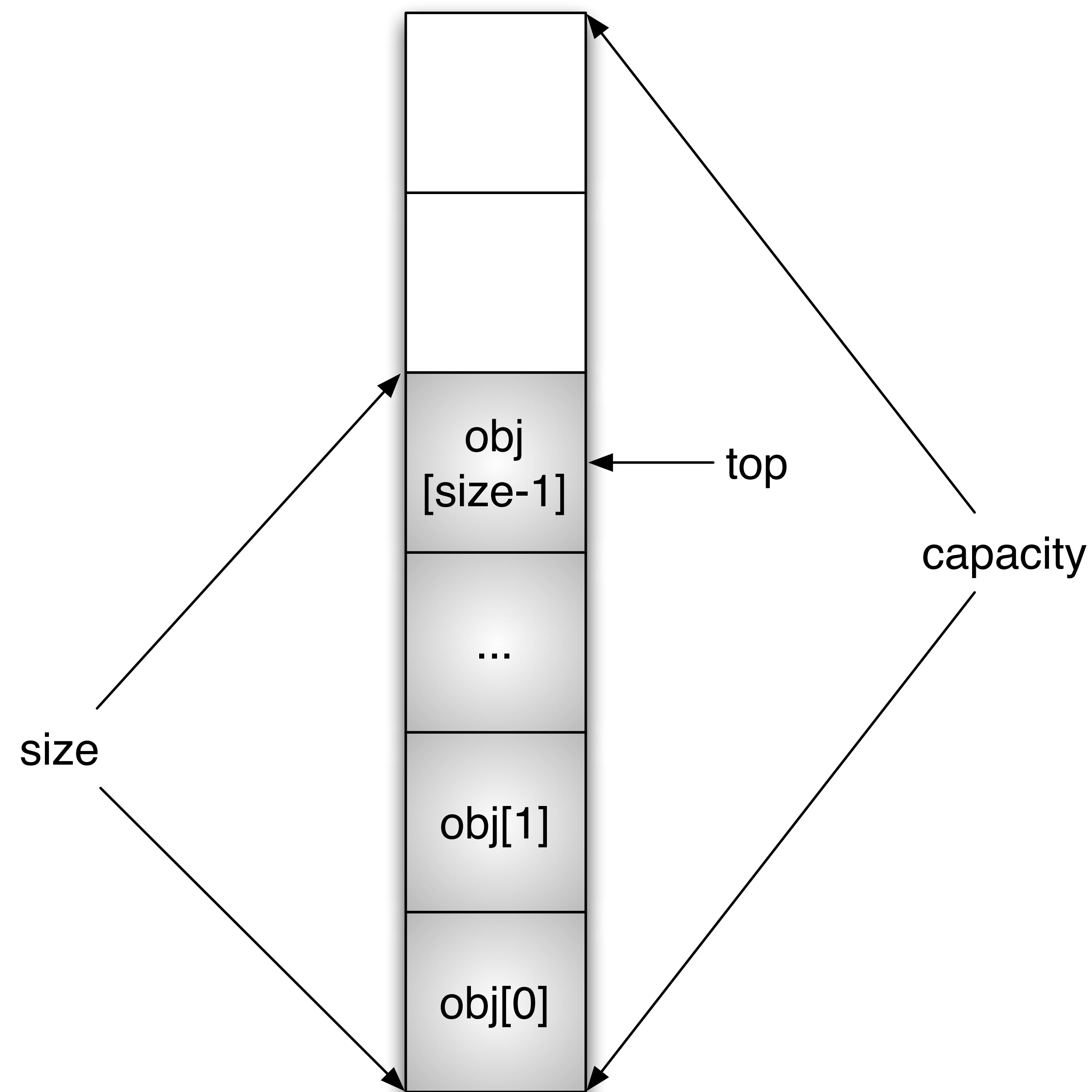
Fraunhofer
FIRST

Implementing a stack



- using an C-array `obj`
- array length defines the *capacity* of the stack
- *size* denotes the number of elements in stack
- `obj[size-1] is the` top element

Implementing a stack




```
struct Stack
{
  value_type* obj;

  size_type  size;

  size_type  capacity;
};

typedef struct Stack Stack;
```

Implementing `push_stack`

```
// Inserts v at the top of the stack.
//
void push_stack(Stack* stack, value_type v)
{
  if (!full_stack(stack))
    stack->obj[stack->size++] = v;
}
```

Fraunhofer FIRST

Implementing `pop_stack`

```
// Removes the element at the top of the stack.
// Requires that the stack is not empty
//
void pop_stack(Stack* stack)
{
  if (!empty_stack(stack))
    --stack->size;
}
```

Fraunhofer
FIRST

Verification of Stack

- implementing a stack is not difficult
- however, you are responsible to verify that your implementation is correct
- functional verification must be based on stack axioms

Fraunhofer
FIRST

From Unit Tests to Unit Proofs

Fraunhofer
FIRST

A Unit Test for Axiom 3

```
void stack_axiom3(Stack* s, Stack* t, value_type v)
{
  CU_ASSERT(equal_stack(s, t));

  push_stack(s, v);
  pop_stack(s);

  CU_ASSERT(equal_stack(s, t));
}
```

Fraunhofer
FIRST

Tests need Test Objects

```
stack_axiom3(Stack* s, Stack* t, value_type v);
```

- must be executed with various values
  - e.g. empty, full, and “normal” stacks
- what is sufficient testing?

Fraunhofer FIRST

Unit Proofs

- proofs can handle very large, even infinite state spaces
- but requires formal specification

Fraunhofer FIRST

Axiom 3 revisited

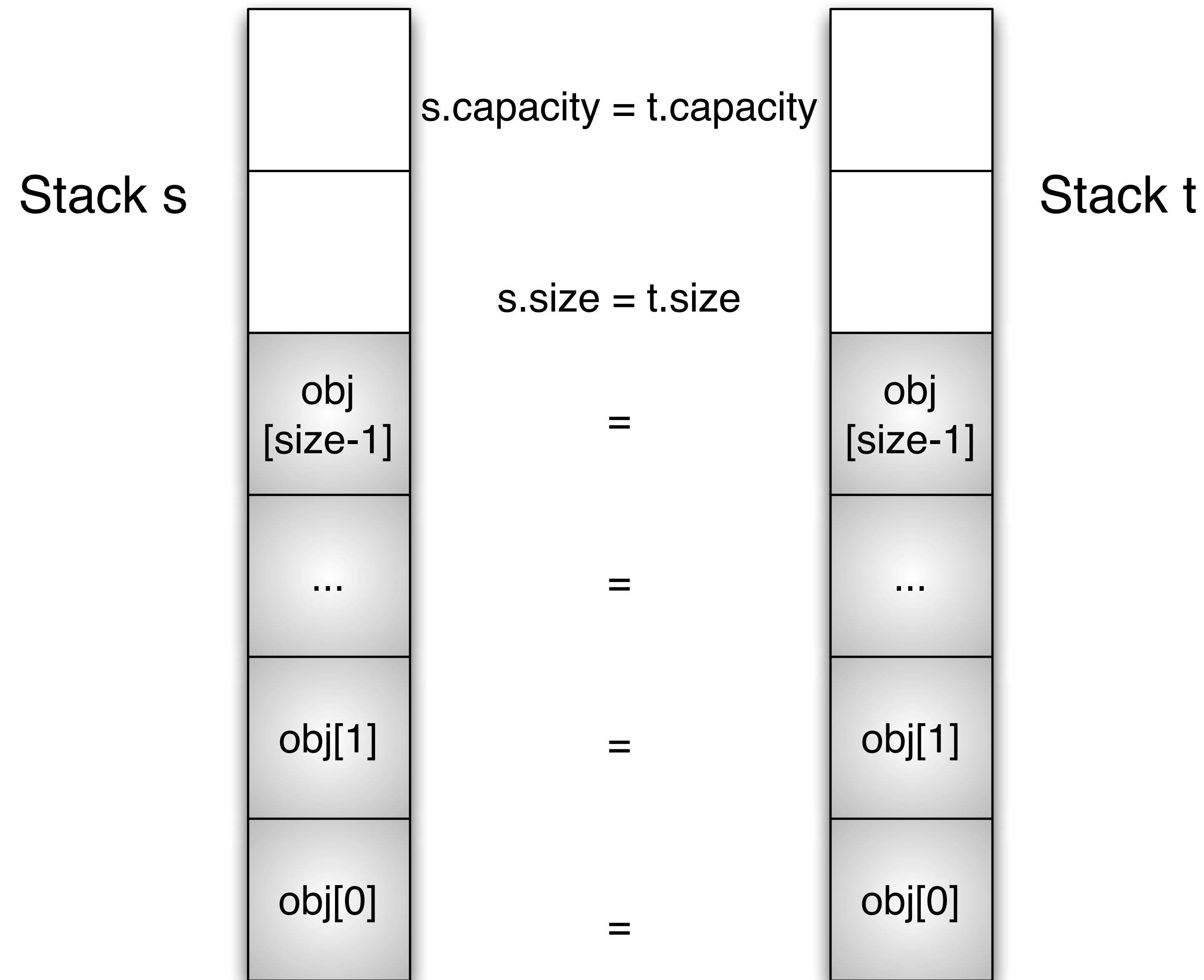
```
void stack_axiom3(Stack* s, Stack* t, value_type v)
{
  CU_ASSERT(equal_stack(s, t));

  push_stack(s, v);
  pop_stack(s);

  CU_ASSERT(equal_stack(s, t));
}
```

Fraunhofer
FIRST

Equality of stacks



Fraunhofer
FIRST

Two equal stacks

Stack s

capacity = 5

size = 3

4	2	17	12	2

Stack t

capacity = 5

size = 3

4	2	17	8	2

Fraunhofer FIRST

Equality of stacks `s` and `t`

```
predicate IsEqualStack(Stack* s, Stack* t) =

    s->capacity == t->capacity &&

    s->size == t->size &&

    \forall integer i;
      0 <= i < s->size ==> s->obj[i] == t->obj[i];
```

Fraunhofer
FIRST

`IsEqualStack` should be

- reflexive: `s == s`
- symmetric: `s == t` ⇒ `t == s`
- transitive:

  (`s == t && t == u`) ⇒ `s == u`

- an *equivalence relation*

Fraunhofer FIRST

Formalizing Reflexivity

```
lemma IsEqualStackReflexive :
  \forall Stack* s; IsEqualStack(s, s);
```

Fraunhofer
FIRST

Formalizing Symmetry

```
lemma IsEqualStackSymmetric :
  \forall Stack *s1, *s2;
     IsEqualStack(s1, s2) ==> IsEqualStack(s2, s1);
```

Formalizing Transitivity

```
lemma IsEqualStackTransitive :
   \forall Stack *s1, *s2, *s3;
      IsEqualStack(s1, s2) && IsEqualStack(s2, s3)
         ==> IsEqualStack(s1, s3);
```

Fraunhofer
FIRST

Automatic Verification of these Lemmas

File Configuration Proof

Proof obligations	Alt-Ergo 0.91	Simplify 1.5.7	Yices 1.0.28 (SS)	CVC3 2.2 (SS)
▽ User goals	✓	✓	✓	✓
Lemma IsEqualStackReflexive	●	●	●	●
Lemma IsEqualStackSymmetric	●	●	●	●
Lemma IsEqualStackTransitive	●	●	●	●

Fraunhofer
FIRST

Using the equality predicate in Axiom 3

```
/*@
    requires IsEqualStack(s, t);

    assigns *s;

    ensures IsEqualStack(s, t);
*/
void stack_axiom3(Stack* s, Stack* t, value_type v)
{
  push_stack(s, v);
  pop_stack(s);
}
```

Fraunhofer
FIRST

Not true for full stacks

```
/*@
    requires IsEqualStack(s, t);

    assigns *s;

    ensures IsEqualStack(s, t);
*/
void stack_axiom3(Stack* s, Stack* t, value_type v)
{
  push_stack(s, v);
  pop_stack(s);
}
```

Fraunhofer
FIRST

A predicate for full stacks

```
predicate
  IsFullStack{L}(Stack* s) =
    (s->size == s->capacity);
```

Fraunhofer FIRST

Excluding full stacks

```
/*@
    requires IsEqualStack(s, t);
    requires !IsFullStack(s);

    assigns *s;

    ensures IsEqualStack(s, t);
*/
void stack_axiom3(Stack* s, Stack* t, value_type v)
{
  push_stack(s, v);
  pop_stack(s);
}
```

Fraunhofer
FIRST

What is a valid stack?

```
predicate IsValidStack{L}(Stack* s)=
  \valid(s) &&
  0 < s->capacity &&
  0 <= s->size <= s->capacity &&
  IsValidRange(s->obj, s->capacity);
```

Fraunhofer
FIRST

Final Form of Axiom 3

```
/*@
    requires IsValidStack(s);
    requires IsValidStack(t);

    requires IsEqualStack(s, t);
    requires !IsFullStack(s);

    assigns *s;

    ensures IsEqualStack(s, t);
*/
void stack_axiom3(Stack* s, Stack* t, value_type v)
{
  push_stack(s, v);
  pop_stack(s);
}
```

Fraunhofer
FIRST

Can we prove Axiom 3?

- no!
- we have not yet formally specified `push_stack` and `pop_stack`

Fraunhofer FIRST

```
/*@
  requires IsValidStack(s);
  ensures  IsValidStack(s);

  behavior not_empty:
    assumes !IsEmptyStack(s);
    assigns s->size;
    ensures s->size == \old(s->size) - 1;
    ensures !IsFullStack(s);

  behavior empty:
    assumes IsEmptyStack(s);
    assigns \nothing;

  complete behaviors;
  disjoint behaviors;
*/
void pop_stack(Stack* s);
```

Fraunhofer
FIRST

Verification of `pop_stack`

Proof obligations	Alt-Ergo 0.92	Simplify 1.5.7	Z3 2.1 (SS)	Yices 1.0.28 (SS)	CVC3 2.2 (SS)	Statistics
User goals	✓	✓		✓	✓	3/3
Function pop_stack Default behavior	✓	✓		✗	✓	3/3
Function pop_stack Normal behavior `empty'	✓	✓		✓	✓	2/2
Function pop_stack Normal behavior `not_empty'	✓	✓		✓	✗	6/6
Function pop_stack Safety	✓	✓		✓	✓	4/4

Fraunhofer
FIRST

```
/*@
  requires IsValidStack(stack);
  ensures  IsValidStack(stack);

  behavior not_full:
    assumes !IsFullStack(stack);
    assigns stack->size;
    assigns stack->obj[stack->size];
    ensures stack->size == \old(stack->size) + 1;
    ensures stack->obj[\old(stack->size)] == v;
    ensures !IsEmptyStack(stack);

  behavior full:
     assumes IsFullStack(stack);
     assigns \nothing;

  complete behaviors;
  disjoint behaviors;
*/
void push_stack(Stack* stack, value_type v);
```

Fraunhofer FIRST

Verification of `push_stack`

Proof obligations	Alt-Ergo 0.92	Simplify 1.5.7	Z3 2.1 (SS)	Yices 1.0.28 (SS)	CVC3 2.2 (SS)	Statistics
User goals	✓	✓		✓	✓	3/3
Function push_stack Default behavior	✓	✓		✗	✓	3/3
Function push_stack Normal behavior `full'	✓	✓		✓	✓	4/4
Function push_stack Normal behavior `not_full'	✗	✓		✗	✗	10/10
Function push_stack Safety	✓	✓		✗	✓	6/6

Fraunhofer
FIRST

Verification of Axiom 3

```
/*@
    requires IsValidStack(s);
    requires IsValidStack(t);

    requires IsEqualStack(s, t);
    requires !IsFullStack(s);

    assigns *s;

    ensures IsEqualStack(s, t);
*/
void stack_axiom3(Stack* s, Stack* t, value_type v)
{
  push_stack(s, v);
  pop_stack(s);
}
```

Fraunhofer
FIRST

Verification of Axiom 3

Proof obligations	Alt-Ergo 0.91	Simplify 1.5.7	Yices 1.0.28 (SS)	CVC3 2.2 (SS)
User goals	✓	✓	✓	✓
Function stack_axiom3 Default behavior	✗	✓	✗	✗
Function stack_axiom3 Safety	✓	✓	✗	✓

Fraunhofer
FIRST

Verification of Axiom 3

We have proved that axiom 3 is satisfied by all valid stacks that are not full

Compare this with unit tests!

Fraunhofer
FIRST

Representation Independence (RI)

Fraunhofer
FIRST

IsEqualStack(s, t)

Stack s

capacity = 5

size = 3

4	2	17	12	2

Stack t

capacity = 5

size = 3

4	2	17	8	2

- not bitwise equal
- two equivalent representatives

Fraunhofer
FIRST

What does RI mean?

Stack s

capacity = 5

size = 3

4	2	17	12	2

Stack t

capacity = 5

size = 3

4	2	17	8	2

- behavior of stack functions must be independent of the representative

Fraunhofer
FIRST

RI is well known to Java Programmers

- consider `java.util.Stack`
- overrides `equal`, `hashCode`, and `toString`
- if `s.equal(t)` then

  ```
  s.hashCode() == t.hashCode()
  ```

  must hold
- RI-violations are a common source of errors in Java

Fraunhofer FIRST

Leibniz Law

Two or more objects are identical if they have all their properties in common.

Fraunhofer FIRST

RI particularly means

```
//@ assert IsEqualStack(s, t);

      push_stack(s, v);
      push_stack(t, v);

//@ assert IsEqualStack(s, t);
```

Fraunhofer
FIRST

More precisely

```
/*@
    requires IsValidStack(s);
    requires IsValidStack(t);
    requires IsEqualStack(s, t);

    assigns *s;
    assigns *t;

    ensures IsEqualStack(s, t);
*/
void push_stack_independence(Stack* s, Stack* t, value_type v)
{
    push_stack(s, v);
    push_stack(t, v);
}
```

Fraunhofer
FIRST

Another Example

```
/*@
    requires IsValidStack(s);
    requires IsValidStack(t);
    requires IsEqualStack(s, t);

    assigns \nothing;

    ensures IsEqualStack(s, t);
    ensures \result == 1;
*/
bool top_stack_independence(Stack* s, Stack* t)
{
    return top_stack(s) == top_stack(t);
}
```

Fraunhofer FIRST

Verification of RI

Proof obligations	Alt-Ergo 0.91	Simplify 1.5.7	Yices 1.0.28 (SS)	CVC3 2.2 (SS)
User goals	✓	✓	✓	✓
Function empty_stack_independe Default behavior	✓	✓	✗	✓
Function empty_stack_independe Safety	✓	✓	✗	✓
Function full_stack_independen Default behavior	✓	✓	✗	✓
Function full_stack_independen Safety	✓	✓	✗	✓
Function pop_stack_independenc Default behavior	✗	✓	✗	✓
Function pop_stack_independenc Safety	✓	✓	✗	✓
Function push_stack_independen Default behavior	✗	✓	✗	✗
Function push_stack_independen Safety	✓	✓	✗	✓
Function top_stack_independenc Default behavior	✓	✓	✗	✓
Function top_stack_independenc Safety	✓	✗	✗	✓

Fraunhofer
FIRST

Conclusions

- not just replacing *unit tests* by *unit proofs*
- a much closer look on equality and representation independence

Fraunhofer FIRST

Questions please!

Fraunhofer FIRST

Implementing `empty_stack`

```
// Returns true if the stack contains no elements,
// and false otherwise
//
bool empty_stack(const Stack* stack)
{
  if (stack->size == 0)
    return 1;
  else
    return 0;
}
```

Fraunhofer
FIRST

Implementing `full_stack`

```
// Returns true if the stack contains
// as many elements as its capacity
// and false otherwise
//
bool full_stack(const Stack* stack)
{
  return (stack->size == stack->capacity) ? 1 : 0;
}
```

Fraunhofer
FIRST

Implementing `top_stack`

```
// Returns the element at the top of the stack
//
value_type top_stack(const Stack* stack)
{
  if (!empty_stack(stack))
    return stack->obj[stack->size - 1];
  else
    return 0;
}
```

Fraunhofer FIRST