ReCrash

Making crashes reproducible by preserving object states

Shay Artzi, Sunghun Kim*, Michael D. Ernst

MIT

* now at HKUST

Eclipse bug 30280:
2 days to reproduce, 4 minutes to fix

2003-01-27 08:01	User: Eclipse crashed… I have no idea why... Here is the stack trace.
2003-01-27 08:26	Developer: What build are you using? Do you have a testcase to reproduce?
2003-01-27 08:39	Developer: Which JDK are you using?
2003-01-28 13:06	User: I’m running Eclipse 2.1, … **I was not able to reproduce the crash.**
2003-01-29 04:33	Developer: Reproduced.
2003-01-29 04:37	Developer: Fixed.

Reproducing crashes

- If a crash can't be reproduced:
  - Hard to fix
  - Hard to validate a solution
- Reproducing a crash is hard!
  - Nondeterminism
  - Configuration and system information
  - Steps to reproduce may be complex or long
  - In-field detection
  - Users rarely provide reproducible bug reports

Approach 1: Postmortem analysis

Examples: stack trace, core dump

Problems:

- Fault (bug) may be far from failure (exception)
  - Faulty method may not be in stack trace
- Too much information
  - Core dump: big; hard to interpret
- Not enough information
  - Shows effects (final values), not causes
  - Need initial values to reproduce the failure

Approach 2: Record & replay

- Logging: record interactions with environment
- Replay: use log to reproduce the execution
- Checkpoint: replay skips part of the execution

Problems:

- Unrealistic overhead
- Invasive changes to HW/OS/application

Record & replay for OO programs

- Object-oriented style uses only nearby state
  - Unit testing depends on this property
- ReCrash reproduces this nearby state
  - Does not replay an execution
  - Static and dynamic analyses reduce the size
- Lightweight: efficient, no harness, usable in-field
- Not guaranteed to reproduce every failure

ReCrash technique

Goal: Convert a crash into a set of unit tests

1. Monitoring: maintain a shadow stack
   - Contains a copy of each method argument
   - On program crash, write the shadow stack to a file
2. Test generation: create many unit tests
   - For each stack frame, create one unit test:
     - Invoke the method using arguments from the shadow stack
     - If the test does not reproduce the crash, discard the test

Maintaining the shadow stack

- On method entry:
  - Push a new shadow stack frame
  - Copy the actual arguments to the shadow stack
- On non-exceptional method exit:
  - Pop the shadow stack frame
- On program failure (top-level exception):
  - Write the shadow stack to a file
    - Serializes all state referenced by the shadow stack

Create one JUnit test per stack frame

Test case for Eclipse bug 30280

Read arguments from the saved shadow stack

Invoke the method from the stack frame

```
public void test_resolveType() {

  AllocExpr rec = (AllocExpr) shadowStack.getArg(0);
  BlockScope arg = (BlockScope) shadowStack.getArg(1);

  rec.resolveType(arg);
}
```

We expect the method to fail as it did at run time

Evaluating unit tests

- Run each generated unit test
- Discard the test if it does not reproduce the run-time exception

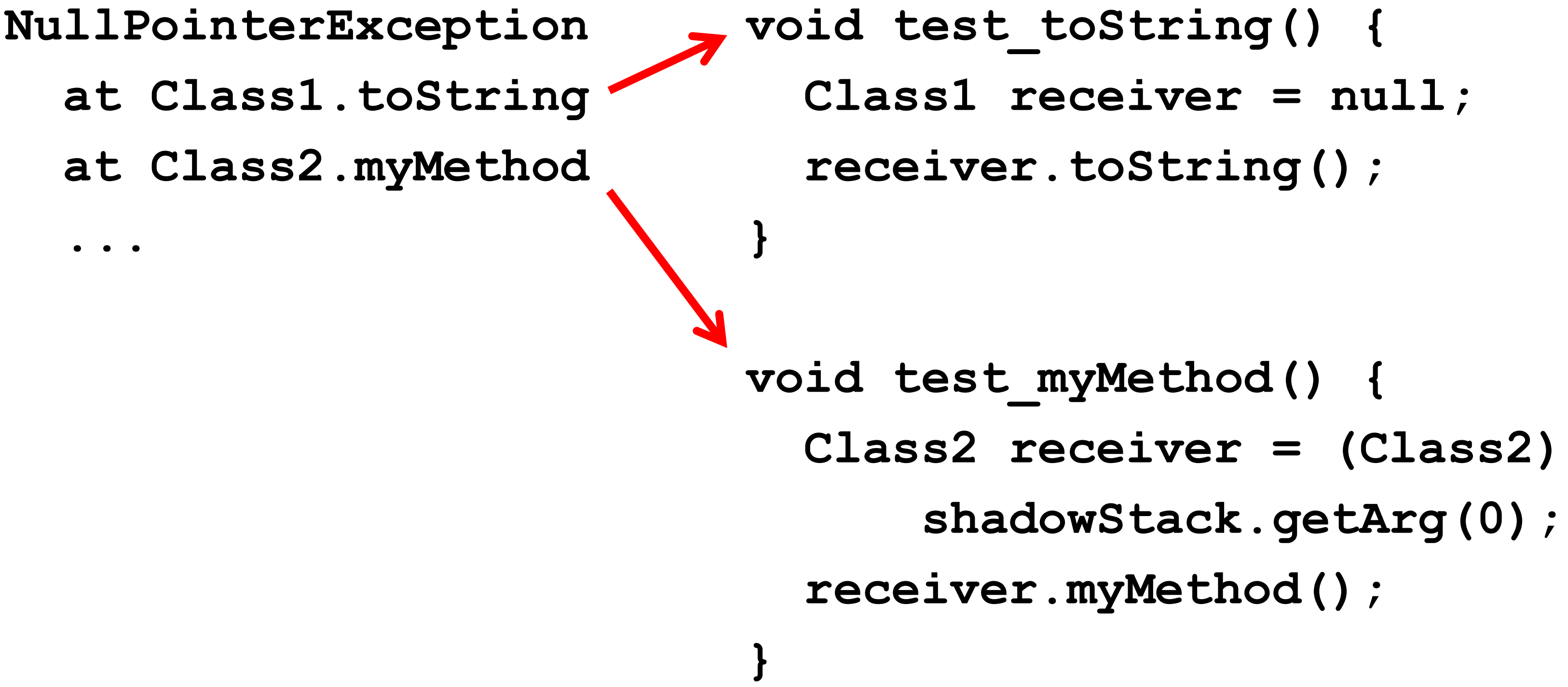
How a developer uses the tests

- In a debugger, step through execution and examine fields
- Experiment by modifying the tests
- Verify a fix
- Create a regression test
  - Replace deserialized objects by real objects or mock objects
  - More readable and robust

Why create multiple tests?

- Not all tests may reproduce the failure
  - Due to state not captured on the shadow stack
    - Sockets, files, nondeterminism, distant program state
    - Does capture all values that are passed as arguments
- Some tests may not be useful for debugging

Not every test is useful



Stack trace:

```
NullPointerException
  at Class1.toString
  at Class2.myMethod
  ...
```

Tests:

```
void test_toString() {
  Class1 receiver = null;
  receiver.toString();
}

void test_myMethod() {
  Class2 receiver = (Class2)
      shadowStack.getArg(0);
  receiver.myMethod();
}
```

Other features of ReCrash

- Non-crashing failures
  - Add a ReCrash annotation
- Caught exceptions that lead to later failures
- Adding extra information to test cases
  - Version number, configuration information
- Reducing the serialized stack
  - Size, privacy

Cost of monitoring

Key cost: copying arguments to shadow stack

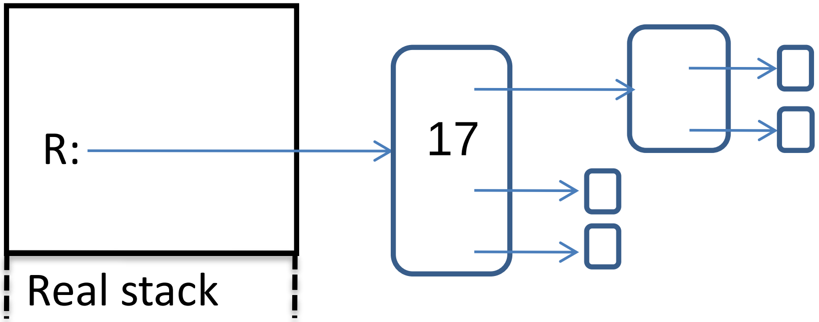
Tradeoff: less information in shadow stack ⇒ lower chance of reproducing failures

1. Depth of copy
   - Deep, reference, or a hybrid
2. Save less information about each argument
   - Focus on important fields
3. Monitor fewer methods
   - Ignore methods not likely to crash or to be useful

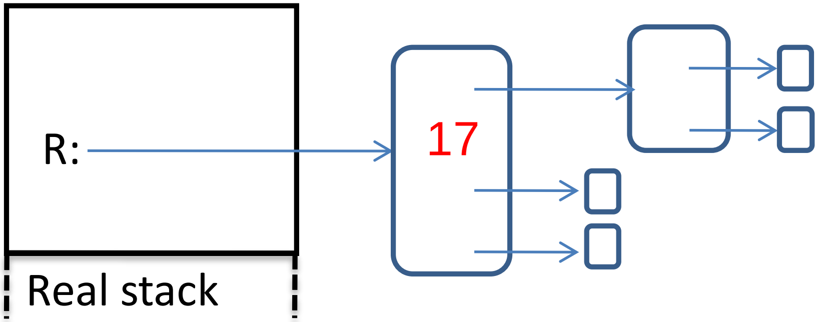
Original program execution

Real stack

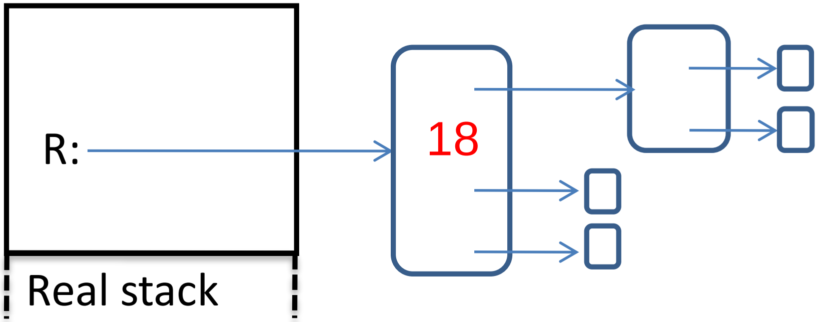
Original program execution



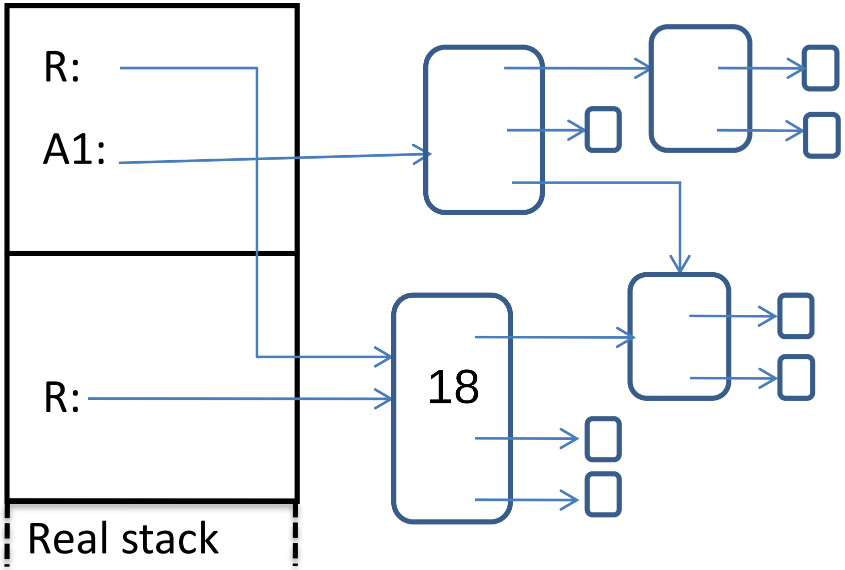
Original program execution



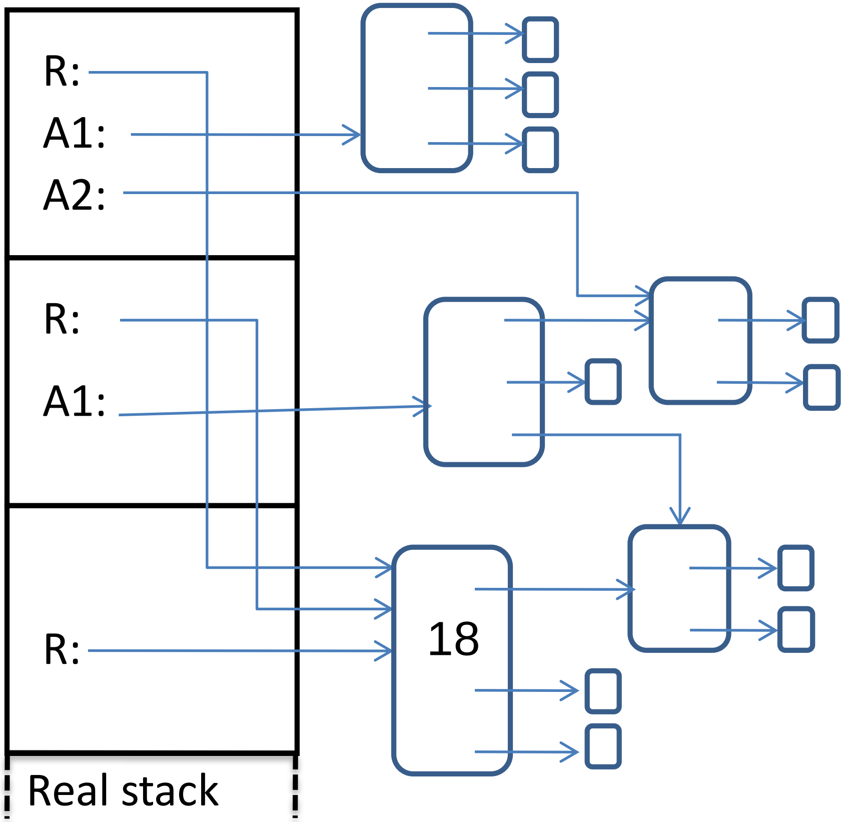
Original program execution



Original program execution



Original program execution



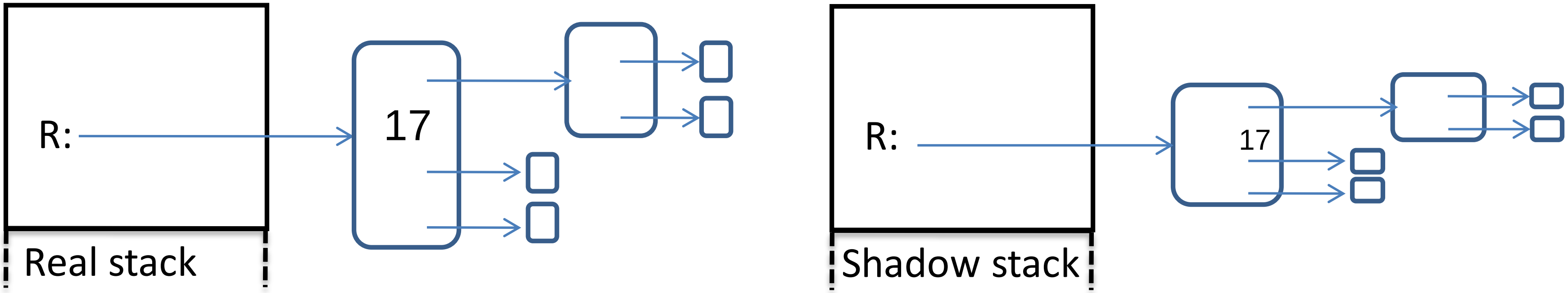
1. Depth of copying

Deep copy

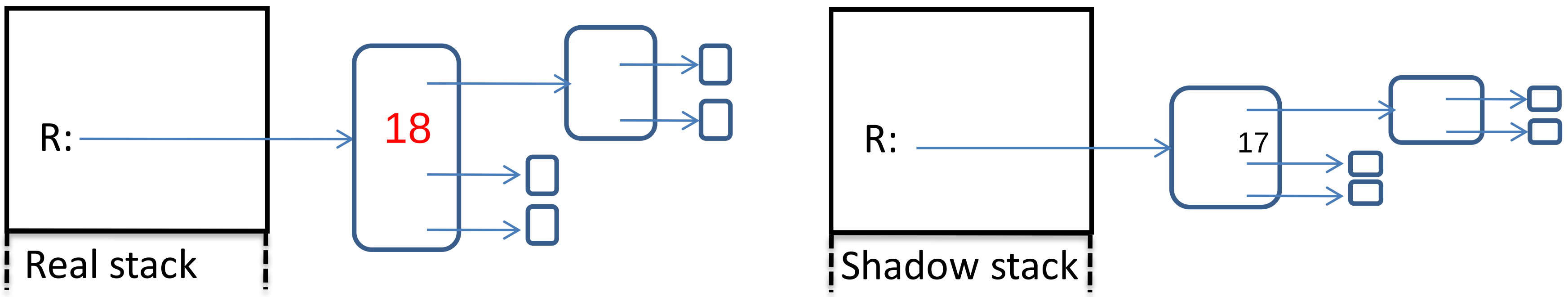
Real stack

Shadow stack

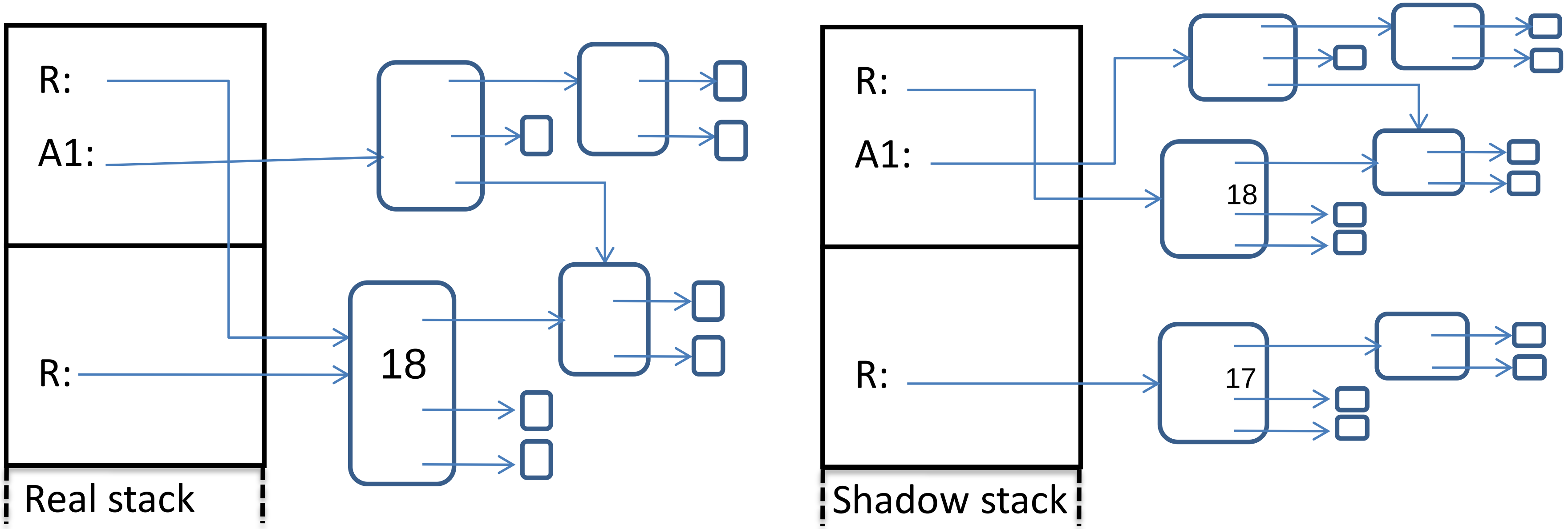
Deep copy



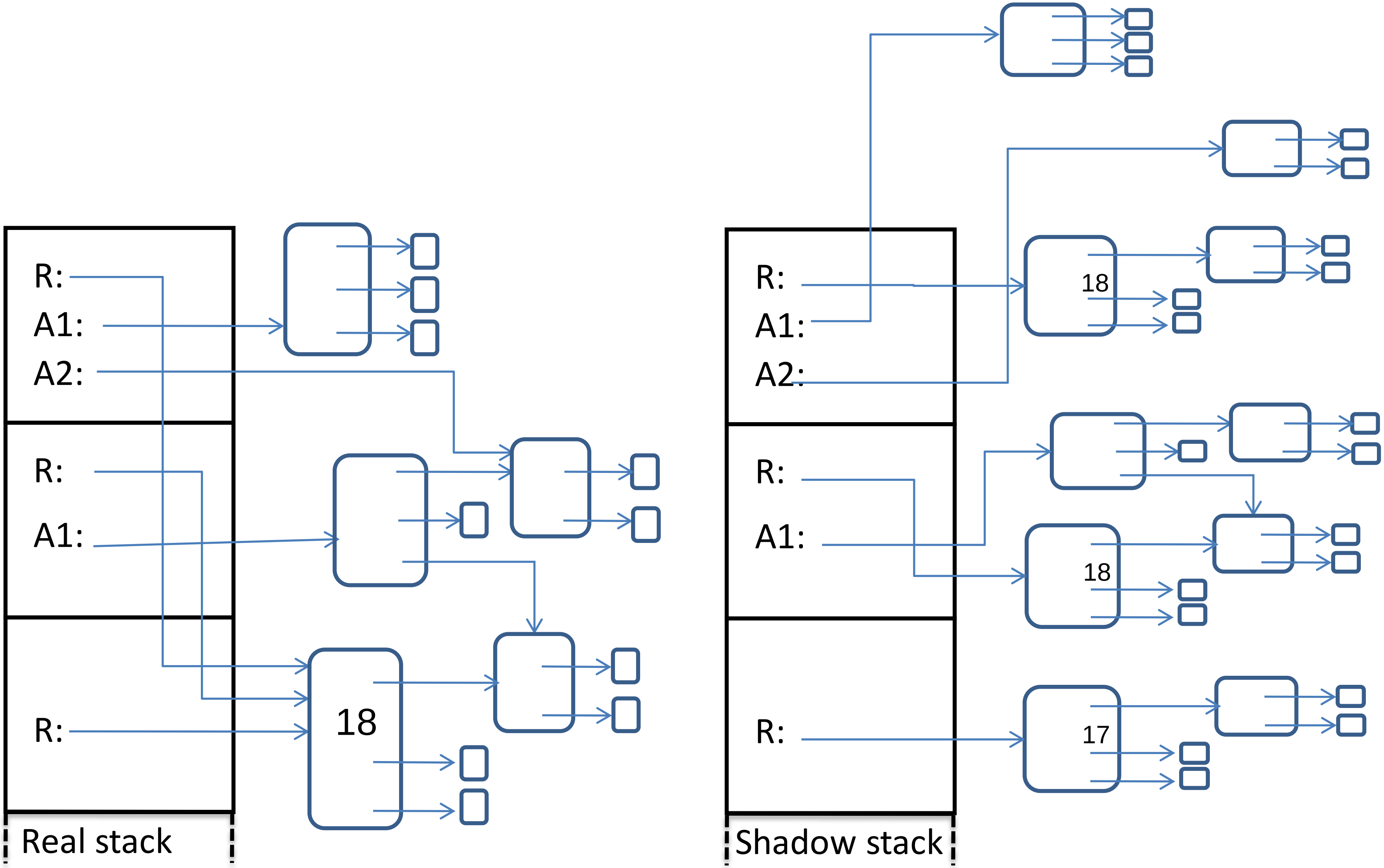
Deep copy



Deep copy



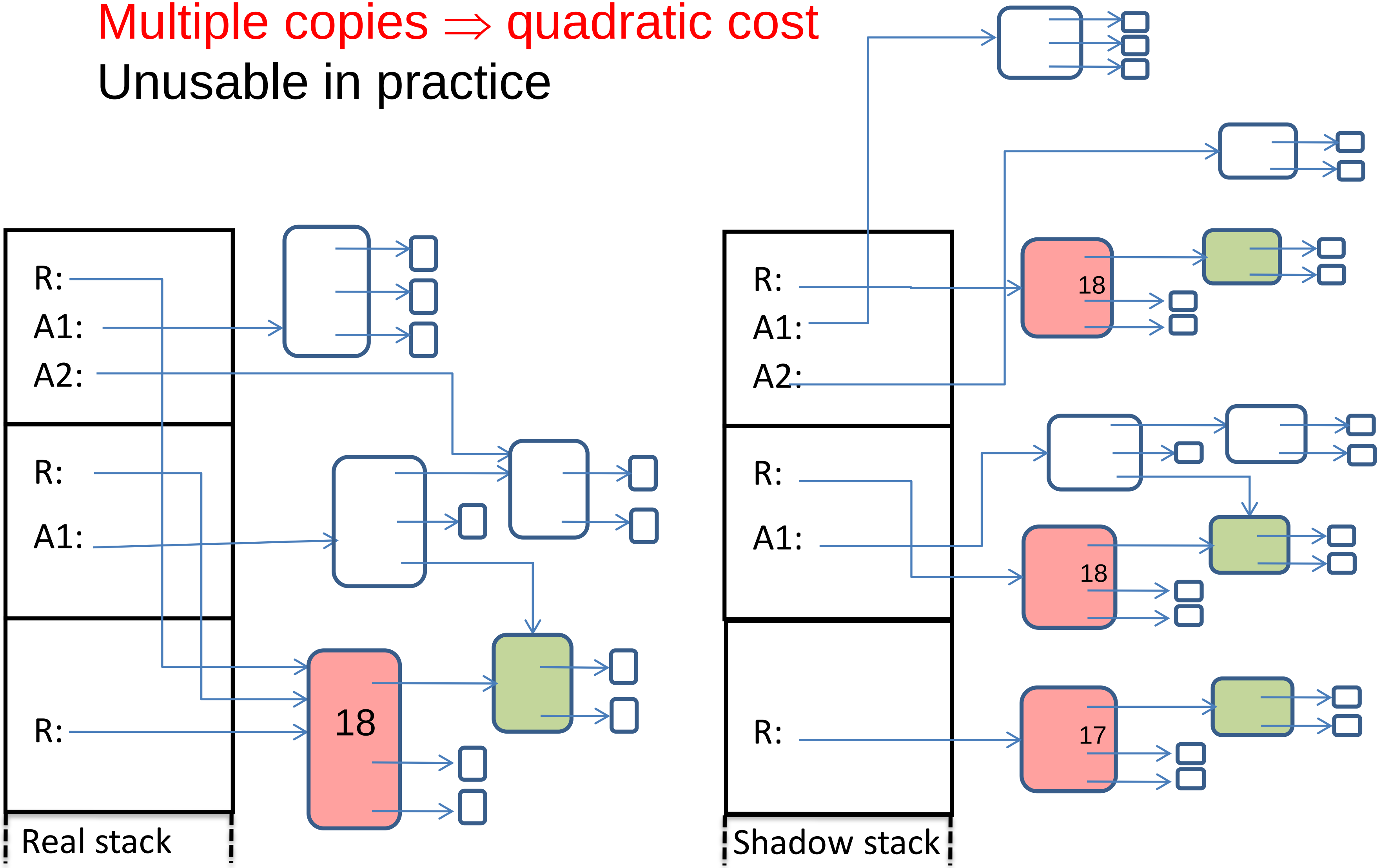
Deep copy



Deep copy

Multiple copies ⇒ quadratic cost

Unusable in practice

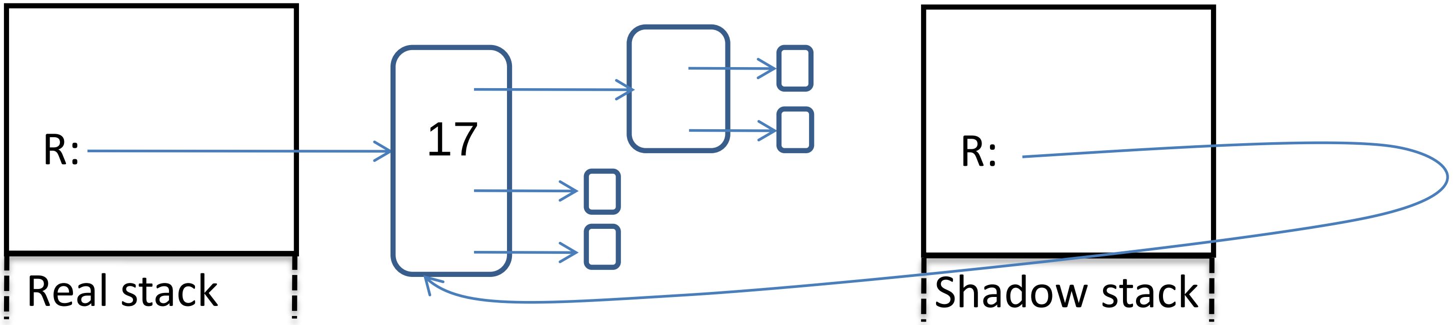


Reference copy

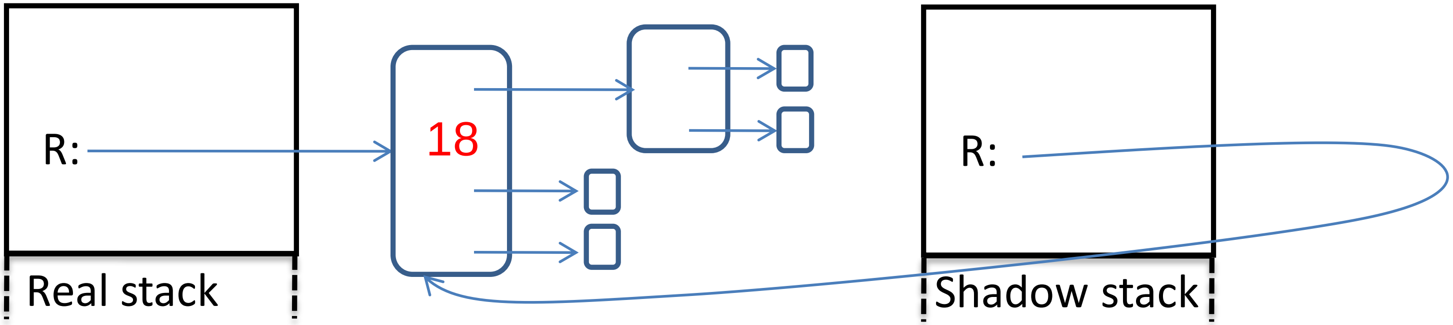
Real stack

Shadow stack

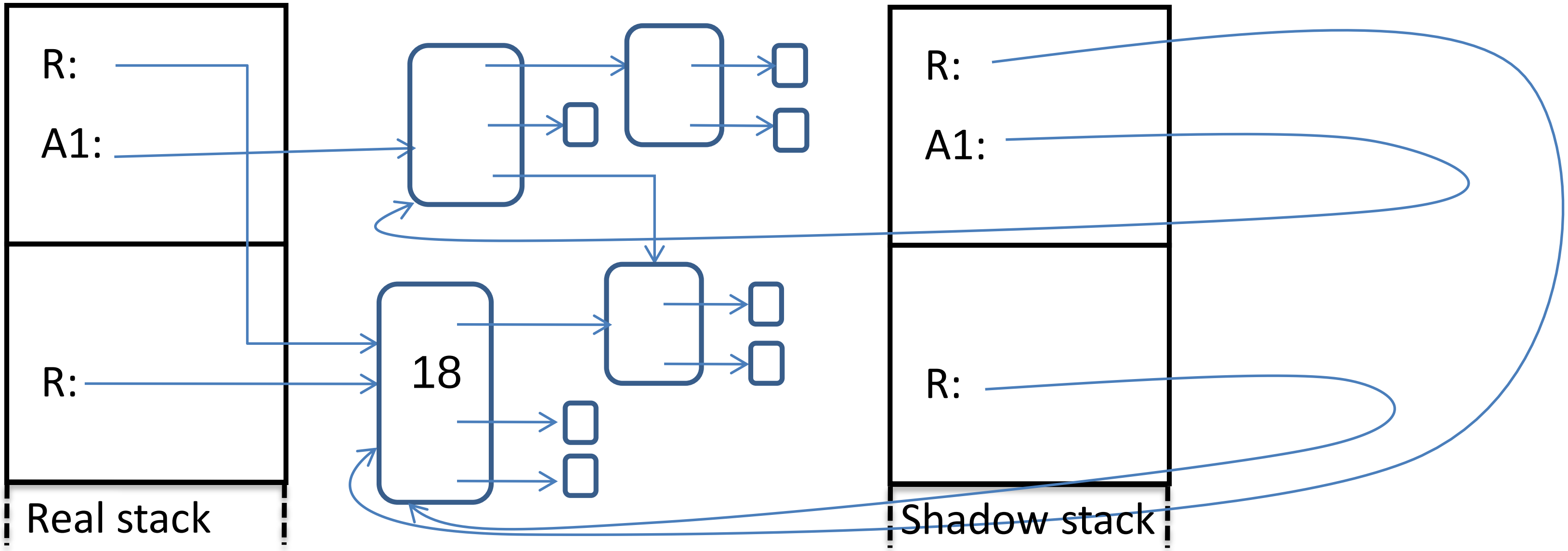
Reference copy



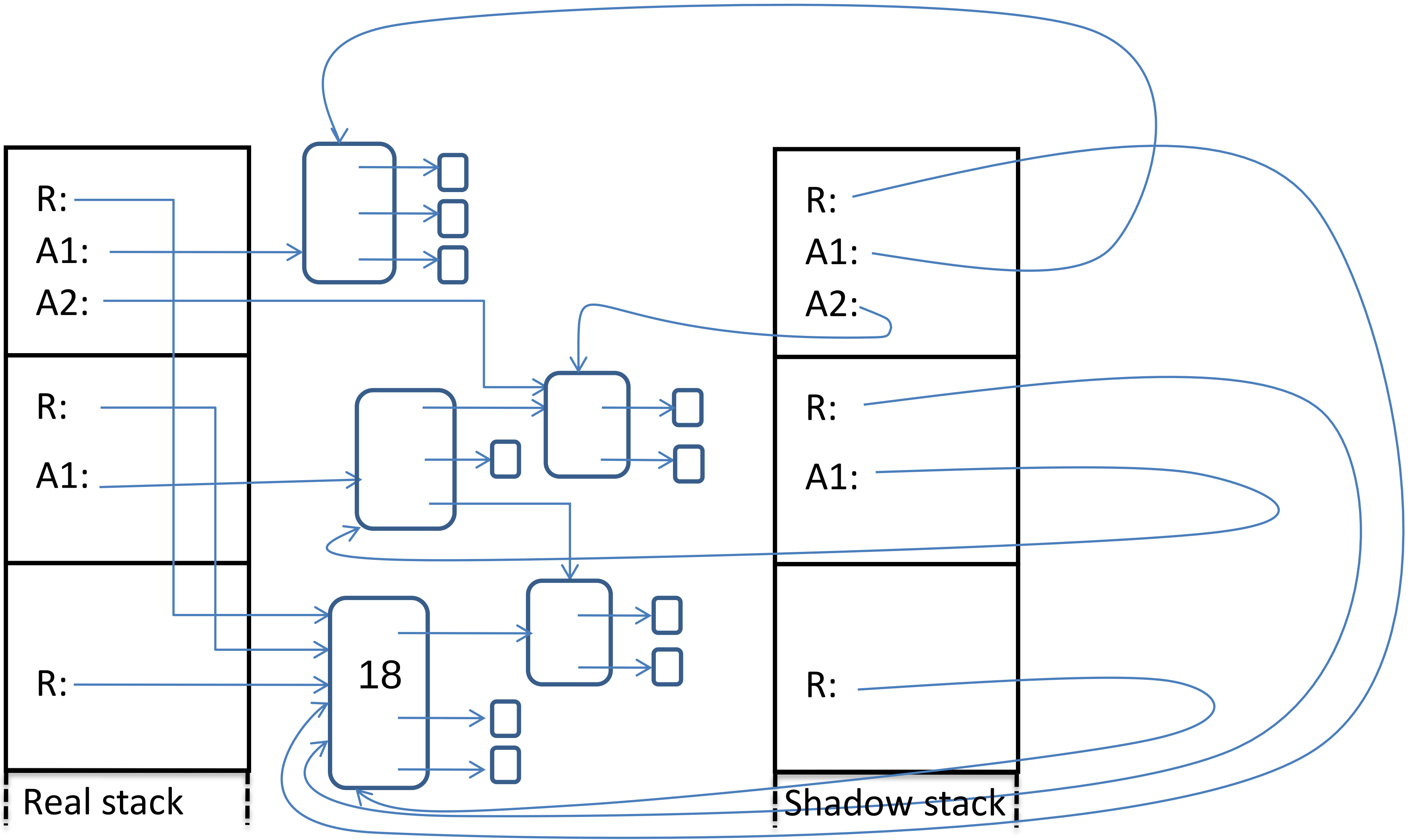
Reference copy



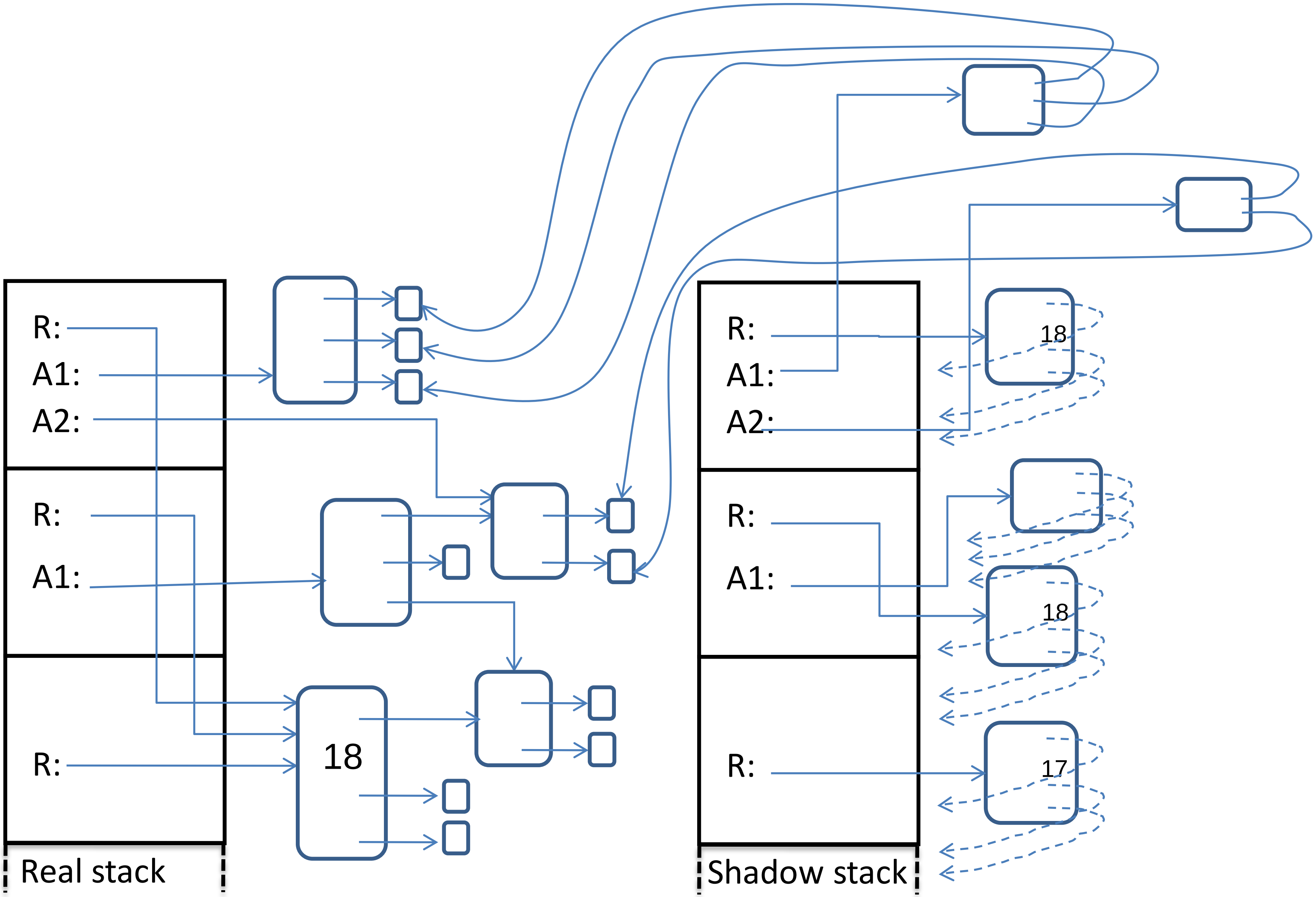
Reference copy



Reference copy

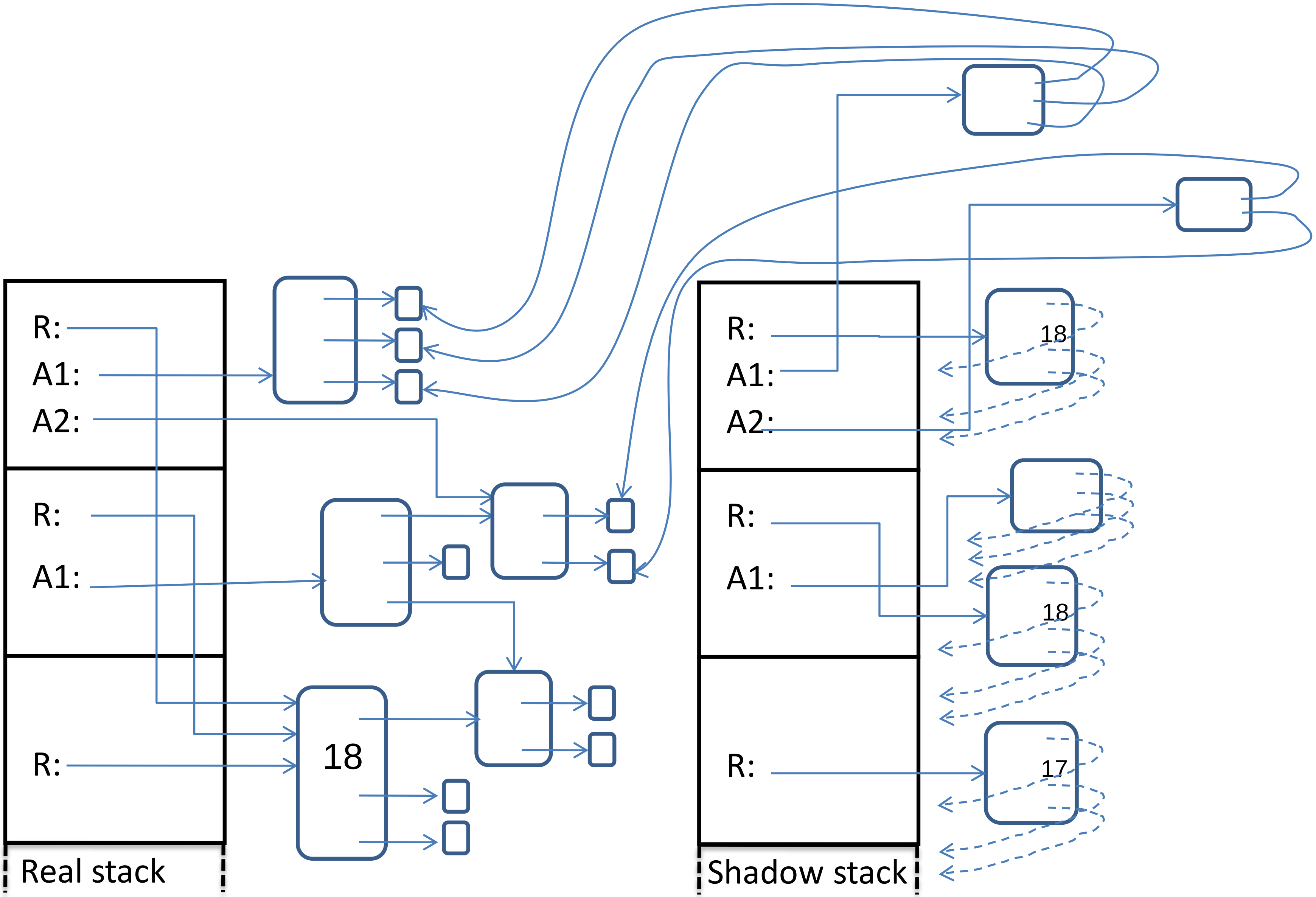


Depth-1 copy

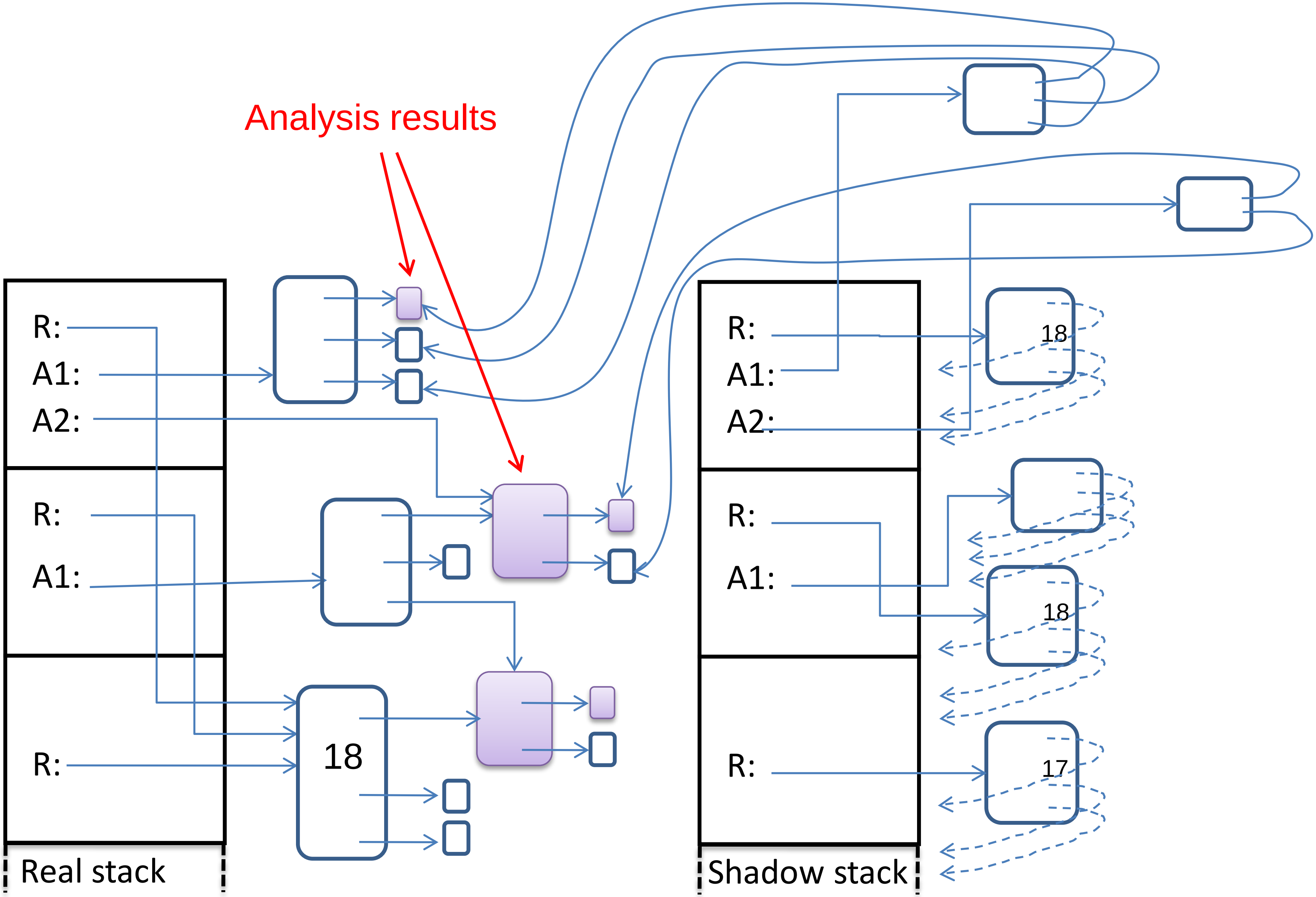


2. Ignoring some fields

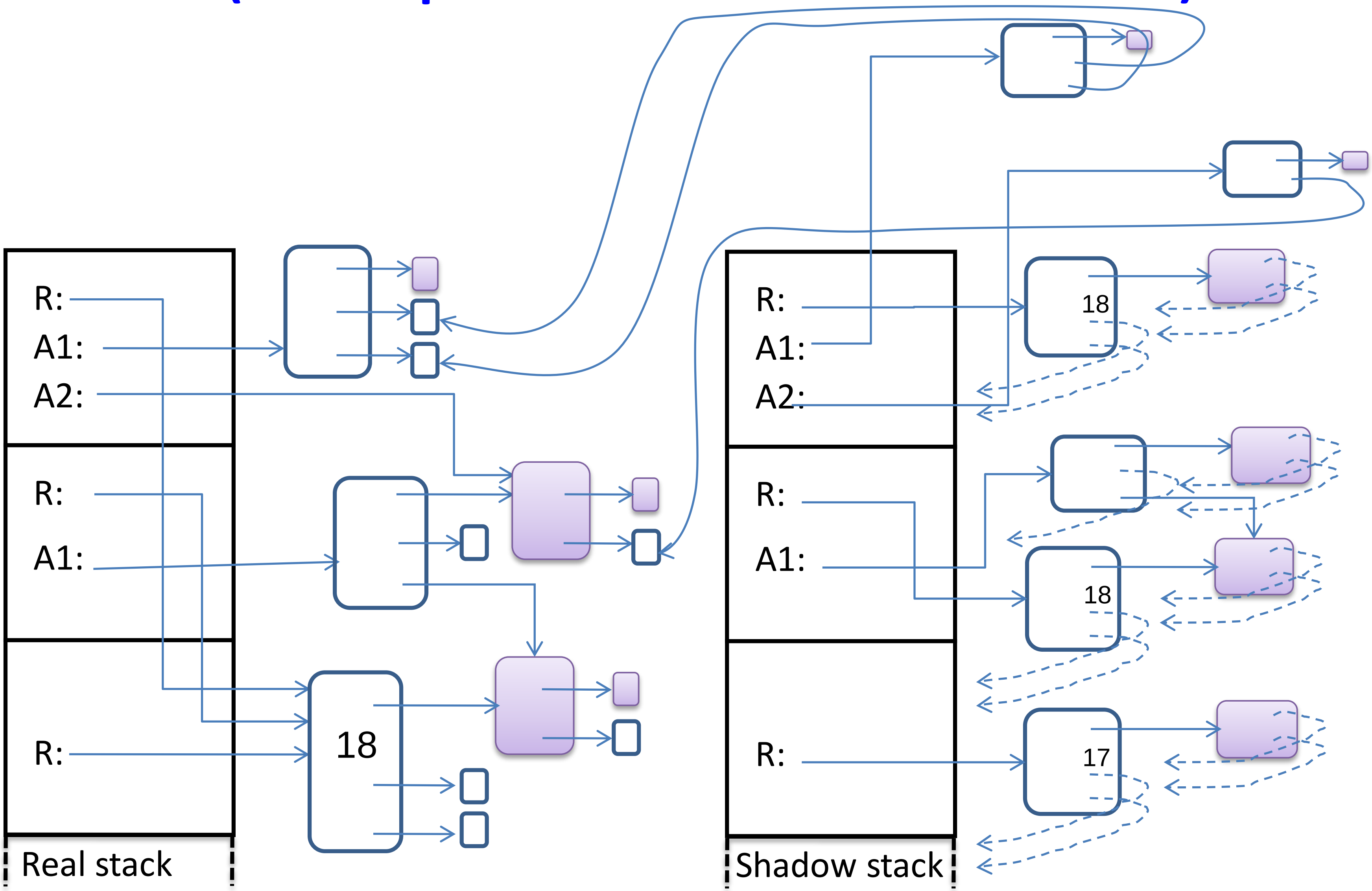
Depth-1 copy



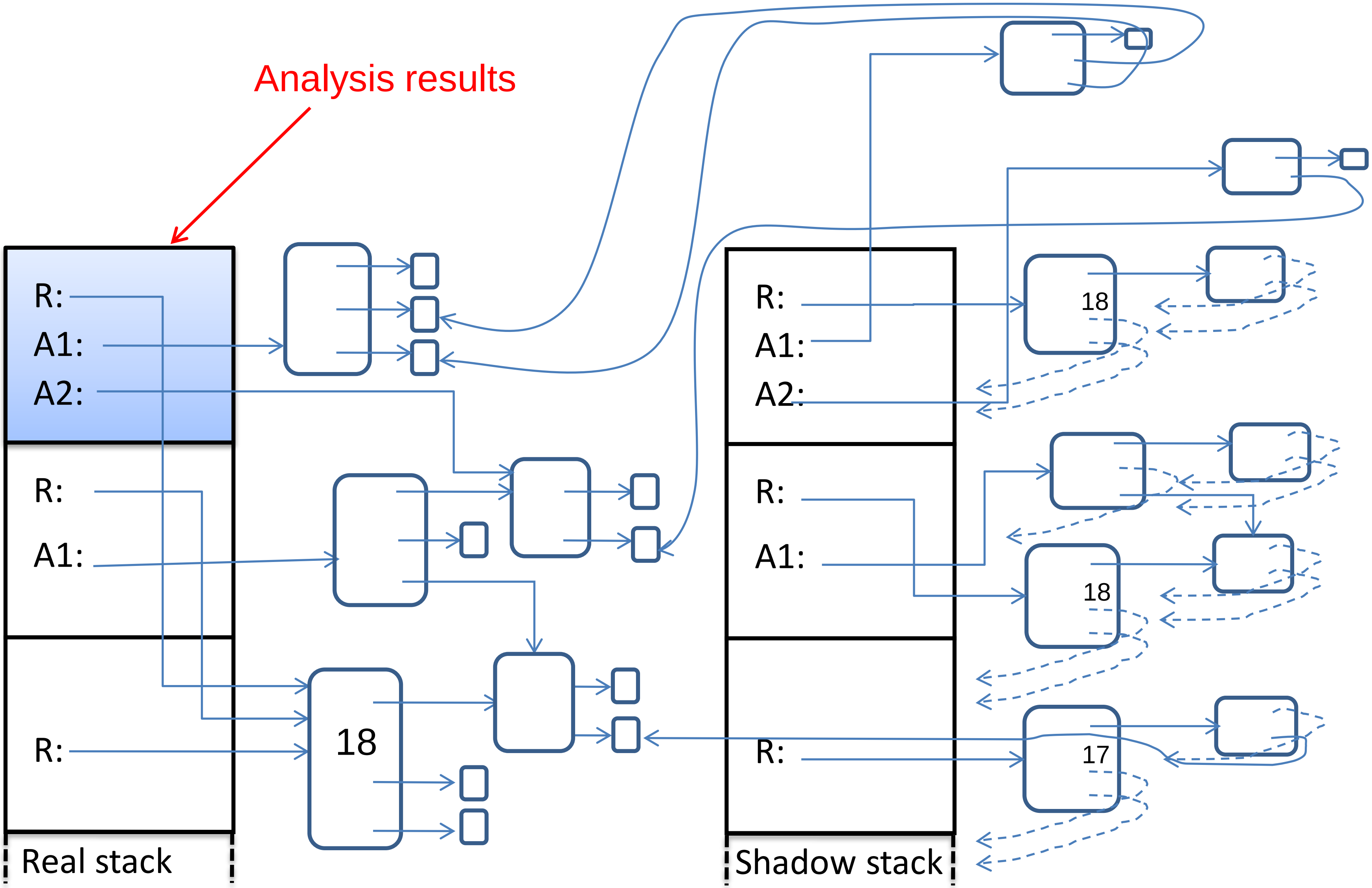
Used fields


Analysis results
R:
A1:
A2:
R:
A1:
R:
18
Real stack
R:
A1:
A2:
R:
A1:
R:
18
18
17
Shadow stack

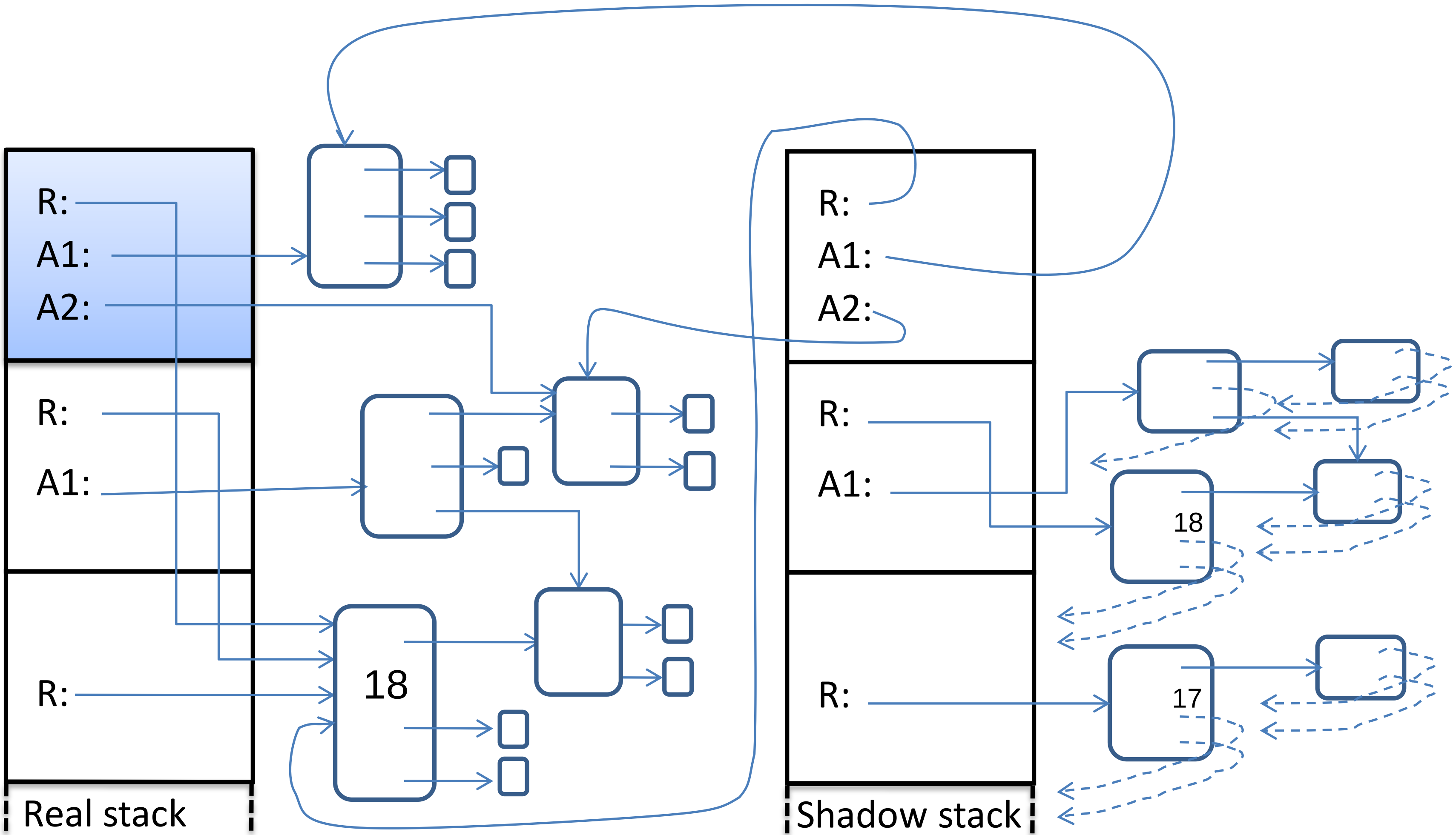
Depth1 + used fields (= Depth2 – unused fields)



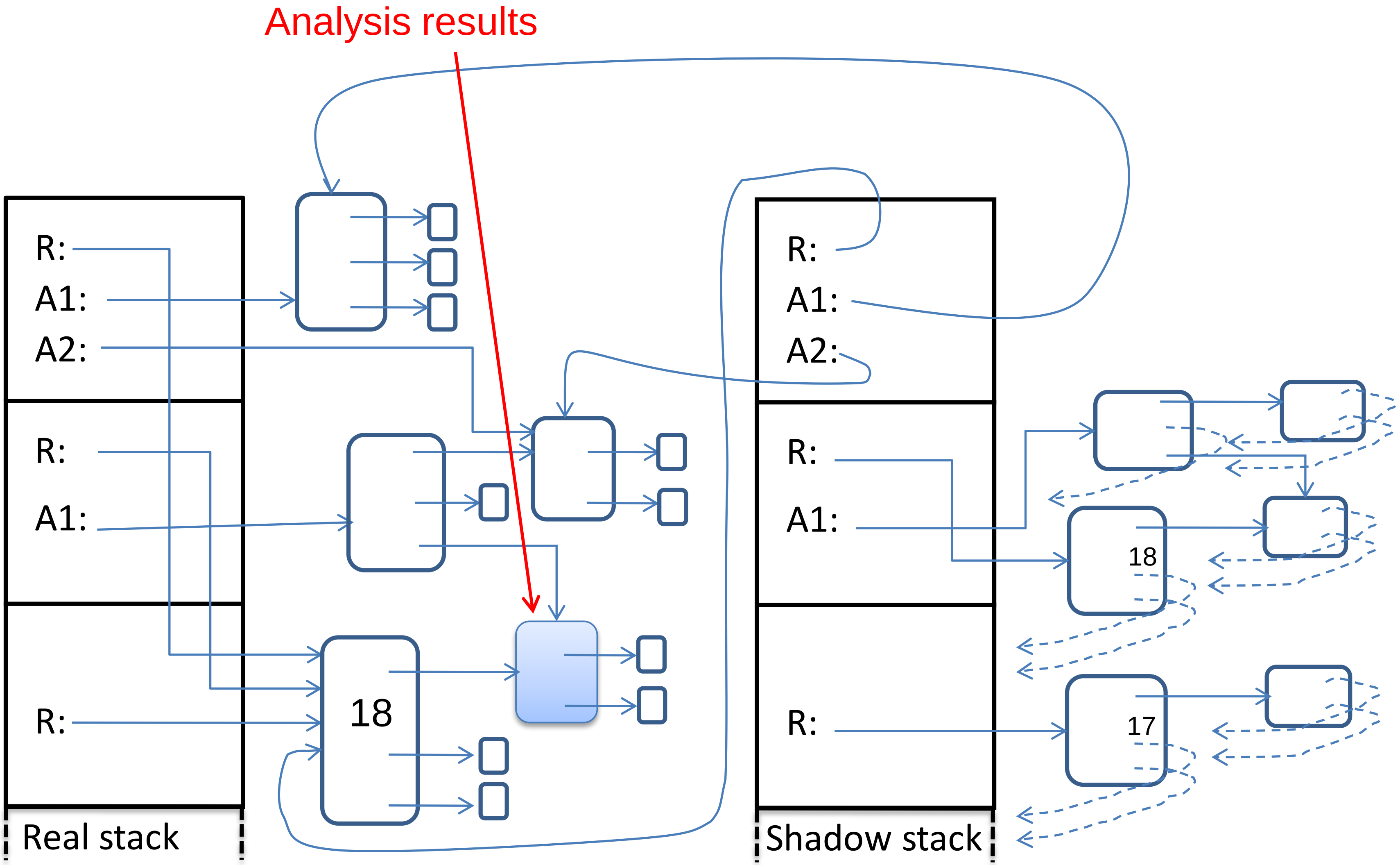
Pure methods



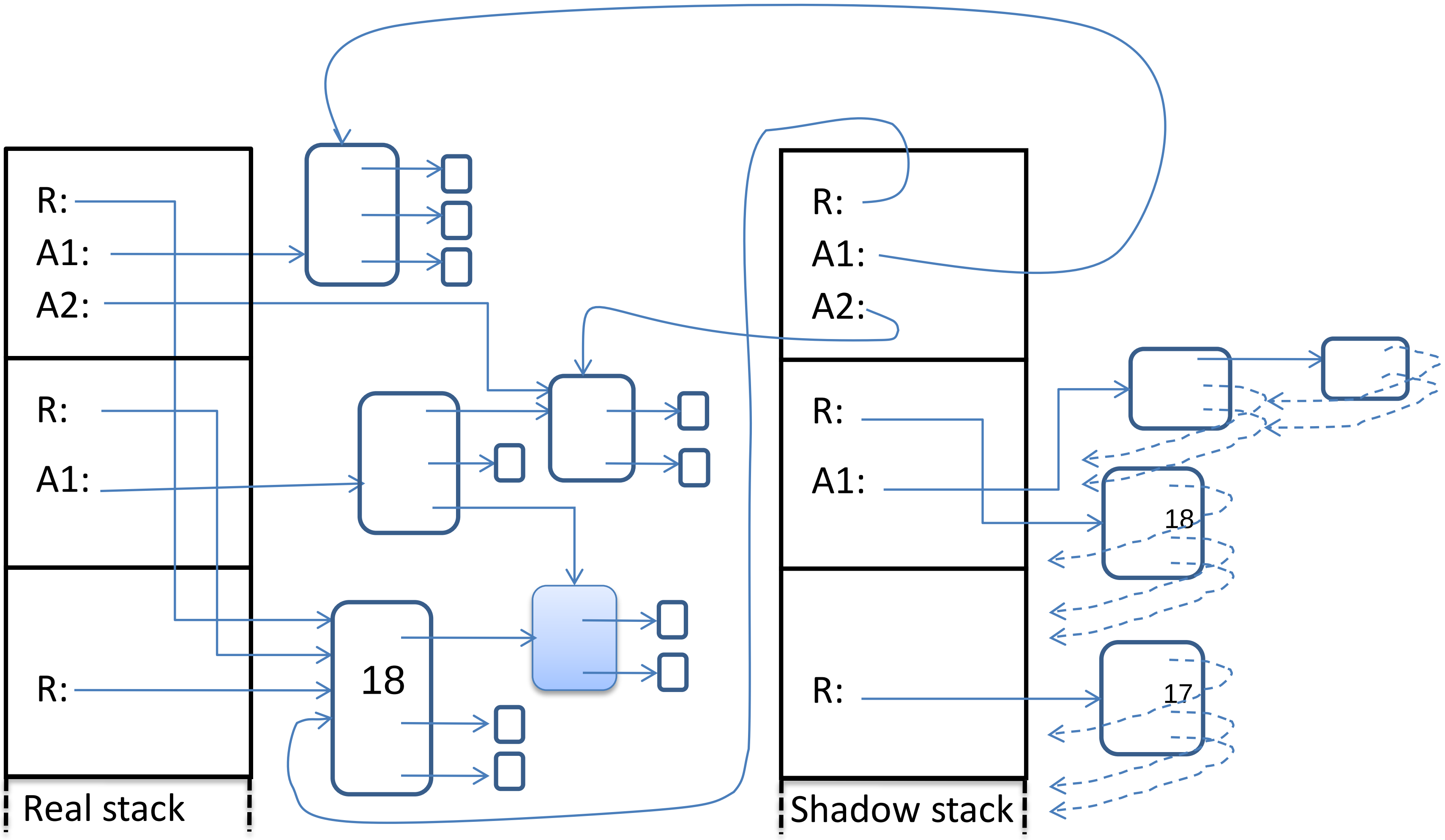
Pure methods



Immutable objects

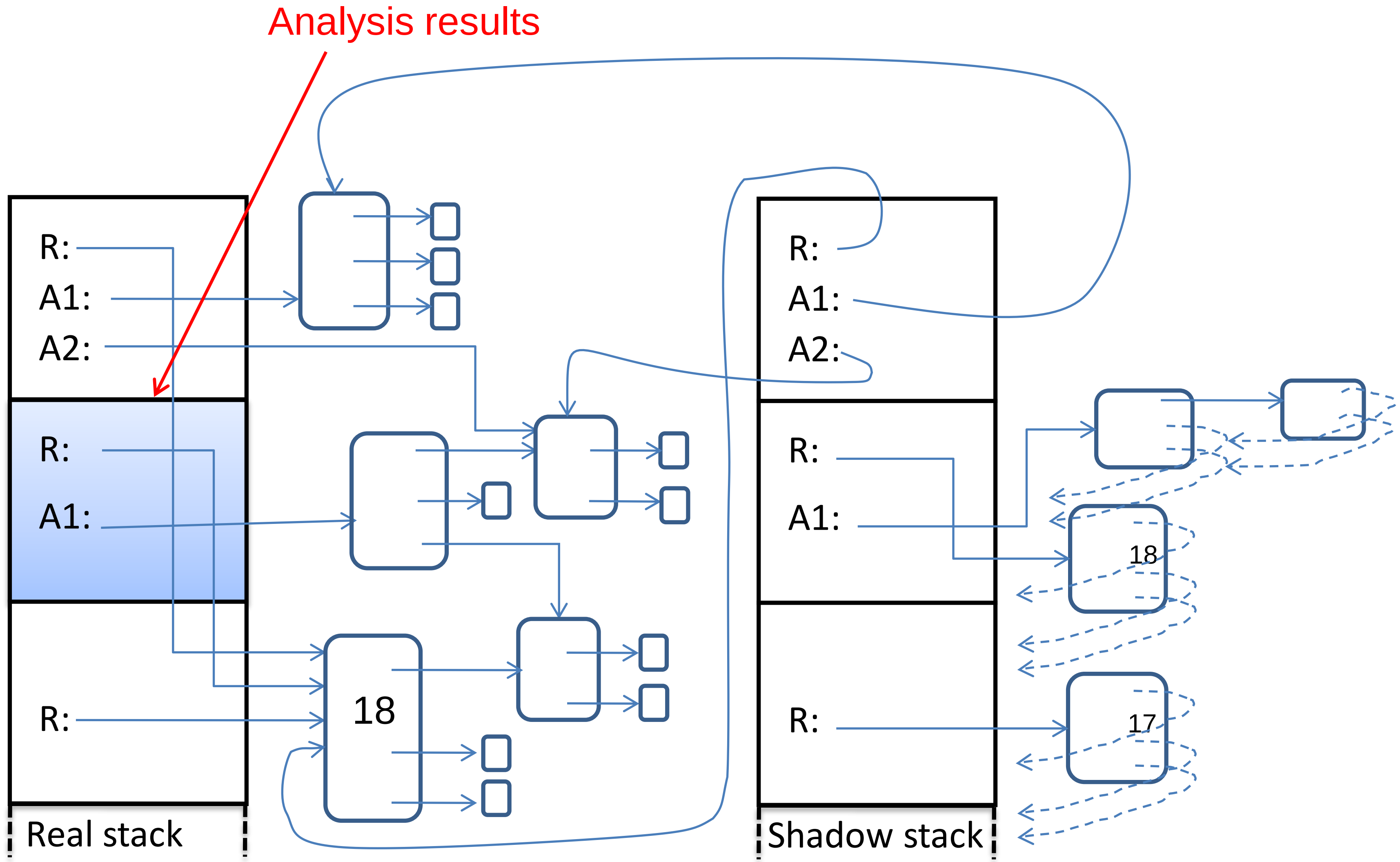


Immutable objects

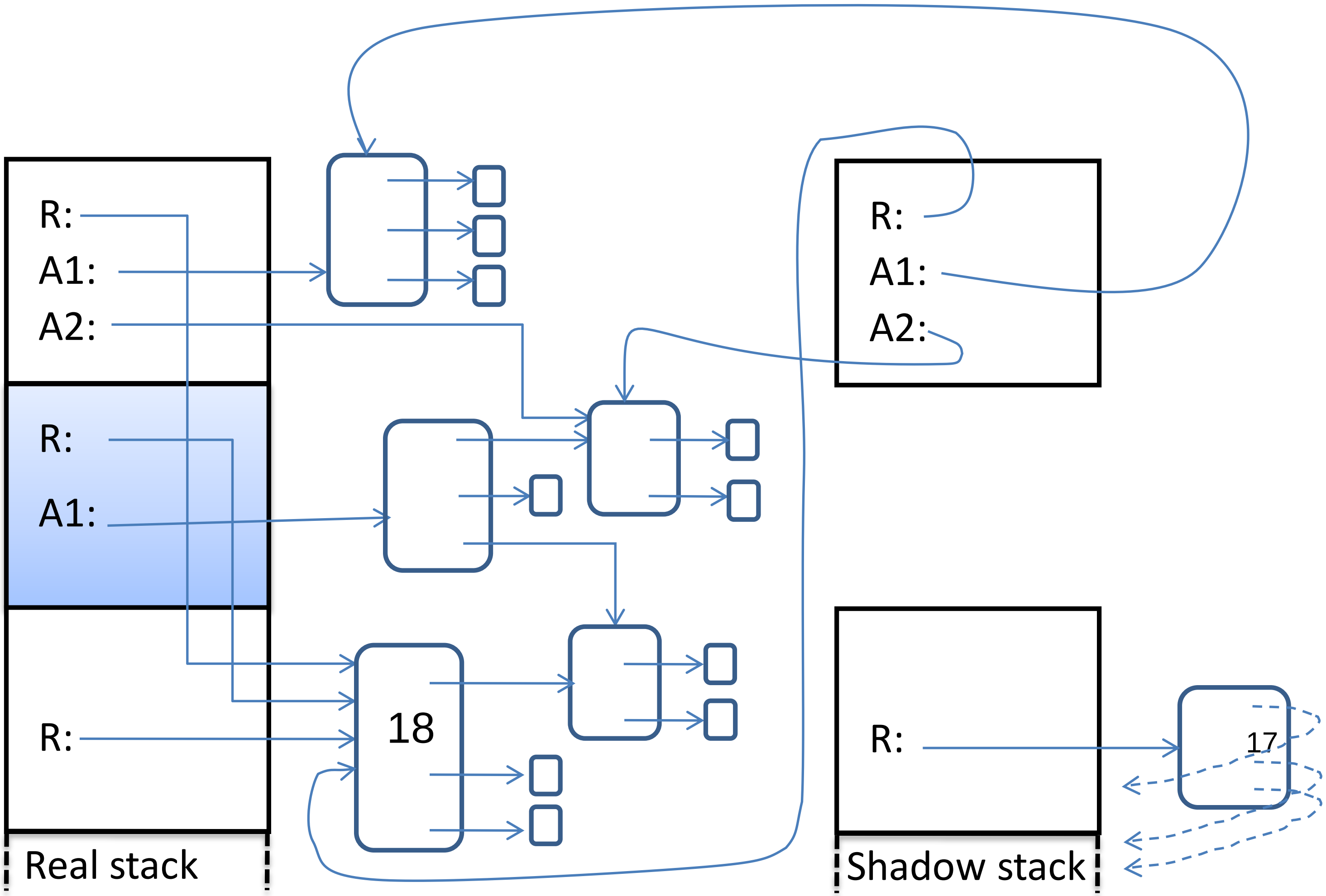


3. Ignoring some methods

Ignored methods



Ignored methods



Methods that are unlikely to be useful

- Trivial methods
- Private methods
- Library methods
- Methods that are unlikely to crash

Second chance mode

Idea: monitor only methods that are likely to crash

- Initially, monitor no methods
- After a crash, add monitoring for methods in the stack trace
  - Can update all clients, not just the one that crashed
- Tradeoffs:
  - + Very low overhead (no overhead until a crash)
  - – Requires a failure to occur twice

Experimental study

1. Can ReCrash reproduce failures?
2. Are the ReCrash-generated tests useful?
3. How large are the test cases?
4. What is the overhead of running ReCrash?

Subject programs

Investigated 11 real crashes from:

- BST: .2 KLOC
- SVNKit: 22 KLOC
- Eclipse compiler: 83 KLOC
- Javac-jsr308: 86 KLOC

Q1: Can ReCrash reproduce failures?

Program	Failure	Candidate tests	Reproducible tests		
			reference copy	depth 1 + used-fields	deep copy
BST	Class cast	3	3	3	3
	Class cast	3	3	3	3
	Unsupported	3	3	3	3
SVNKit	Index bounds	3	3	3	3
	Null pointer	2	2	2	2
	Null pointer	2	2	2	2
Eclipsec	Null pointer	13	0	1	8
Javac-jsr308	Null pointer	17	5	5	5
	Illegal arg	23	11	11	11
	Null pointer	8	1	1	1
	Index bounds	28	11	11	11

Q1: Can ReCrash reproduce failures?

Program	Failure	Candidate tests	Reproducible tests		
			reference copy	depth 1 + used-fields	deep copy
BST	Class cast	3	3	3	3
	Class cast	3	3	3	3
	Unsupported	3	3	3	3
SVNKit	Index bounds	3	3	3	3
	Null pointer	2	2	2	2
	Null pointer	2	2	2	2
Eclipsec	Null pointer	13	0	1	8
Javac-jsr308	Null pointer	17	5	5	5
	Illegal arg	23	11	11	11
	Null pointer	8	1	1	1
	Index bounds	28	11	11	11

Q2: Are the ReCrash tests useful?

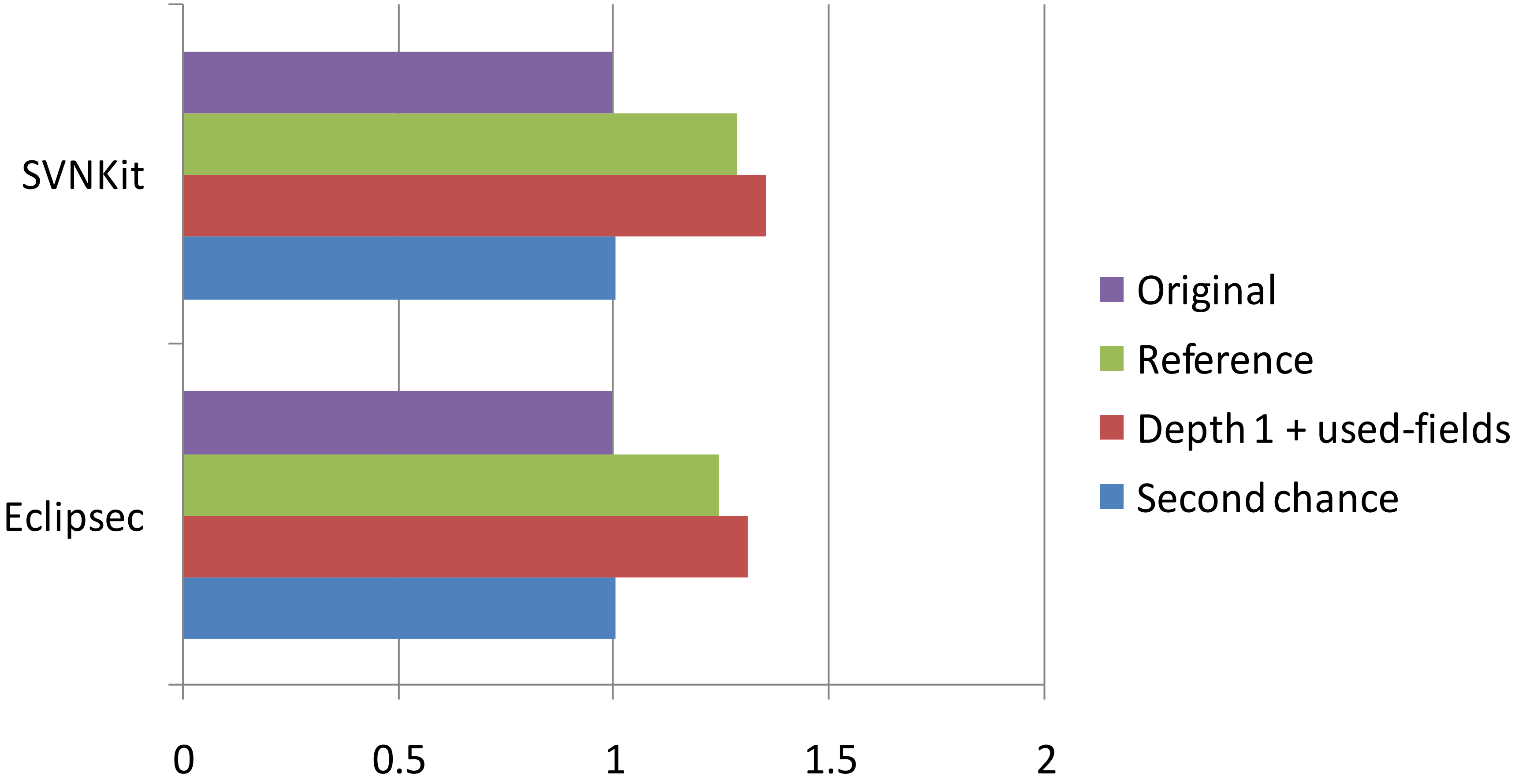
- Developers found the tests useful
  - Developer 1: “You don’t have to wait for the crash to occur again”; also liked multiple tests
  - Developer 2: “Using ReCrash, I was able to jump (almost directly) to the necessary breakpoint”
- Developers found the stack trace insufficient
  - Unable to reproduce
  - The failure may be far removed from the fault

Q3: How large are the test cases?

- The JUnit test suite uses the shadow stack
- Serializes all reachable parts of the heap

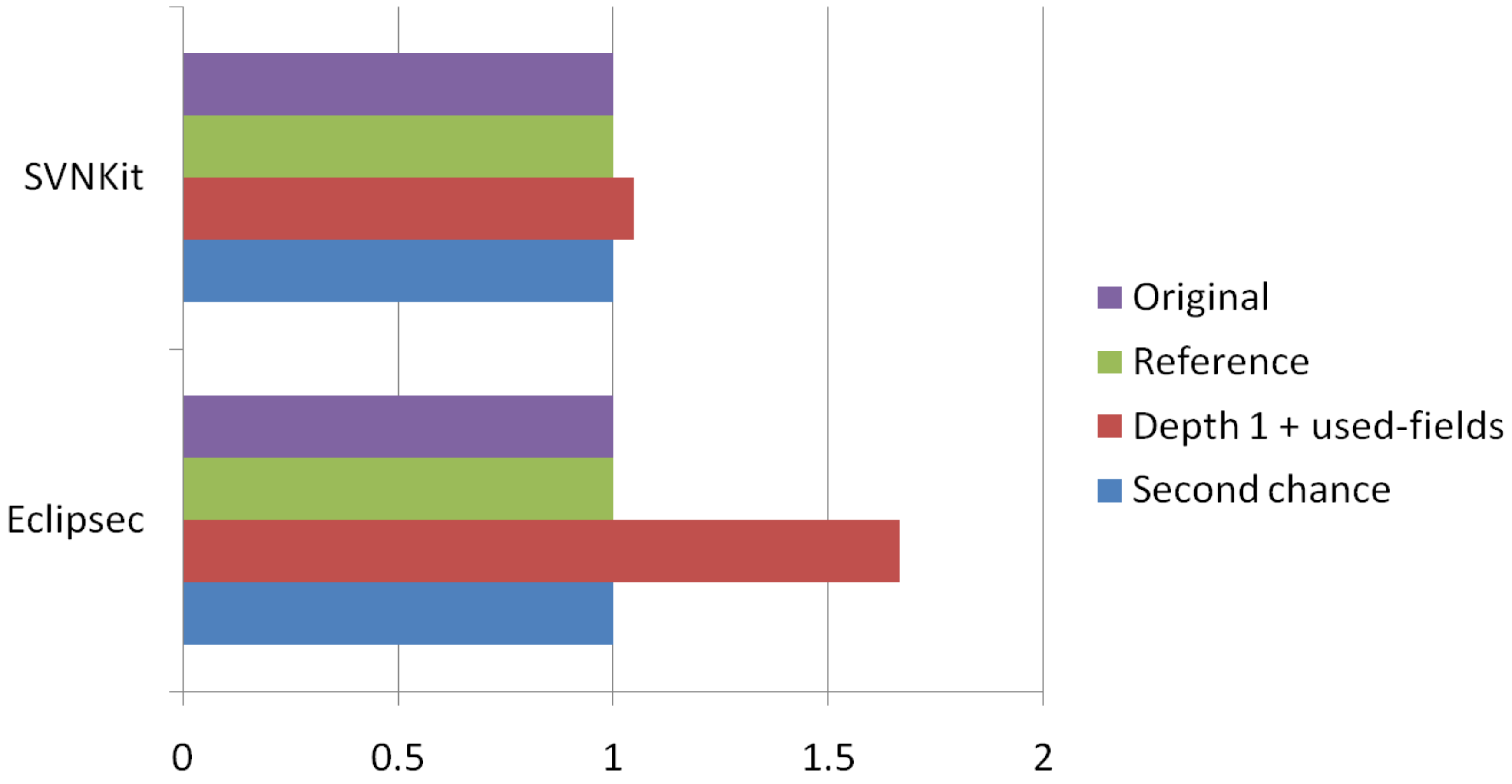
Program	Average shadow stack size (KB)
BST	12
SVNKit	34
Eclipsec	62
Javac-jsr308	422

Q4: Time overhead of ReCrash



Overhead of instrumented program in the field

Q4: Memory overhead of ReCrash



Absolute memory overhead: .2M – 4.7 M

Generating unit tests from system runs

- Test factoring [Saff 2005, Elbaum 2006]
  - Developer selects a portion of the program
  - System logs interactions with the environment
  - Unit test replays execution in a test harness
- Contract-driven development [Leitner 2007]
  - Reference copying, intended for durable tests
- Backward-in-time debuggers [Lienhard 2008]
  - Heavier-weight logging and checkpoints

Future work

- Capture more state
  - Concurrency, timing, external resources
- Other implementation tradeoffs
  - Copy-on-write
  - Existing VM hooks
  - Logging/debugging techniques
  - These are probably orthogonal to ReCrash

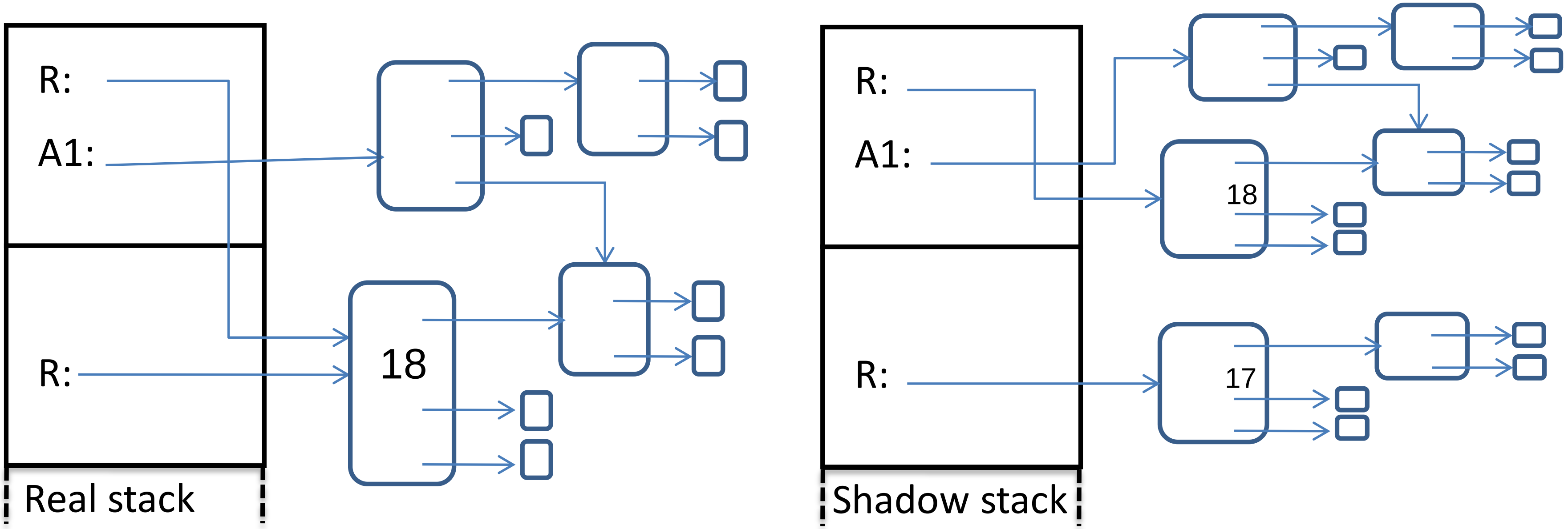
ReCrash converts failures into tests

- ReCrash effectively reproduces failures
  - Replicates program states
  - Generates multiple unit tests
- The unit tests are useful
- Low overhead
  - Records only relevant parts of an execution
  - 4 program analyses; second chance mode
  - Can deploy instrumented programs in the field
- Download: http://pag.csail.mit.edu/ReCrash/

ReCrash converts failures into tests

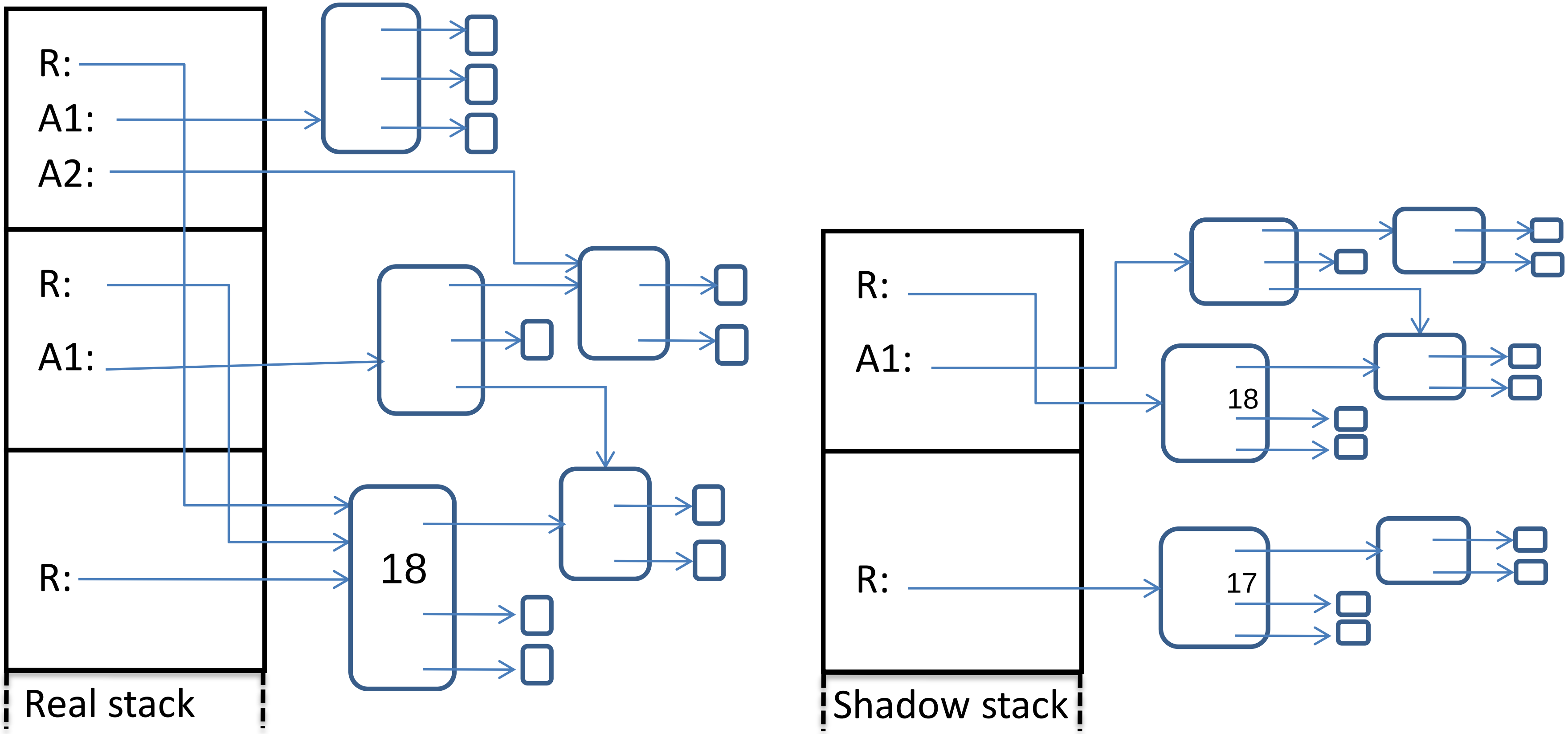
- ReCrash effectively reproduces failures
  - Replicates program states
  - Generates multiple unit tests
- The unit tests are useful
- Low overhead
  - Records only relevant parts of an execution
  - 4 program analyses; second chance mode
  - Can deploy instrumented programs in the field
- Download: http://pag.csail.mit.edu/ReCrash/

Maintaining the shadow stack



Maintaining the shadow stack

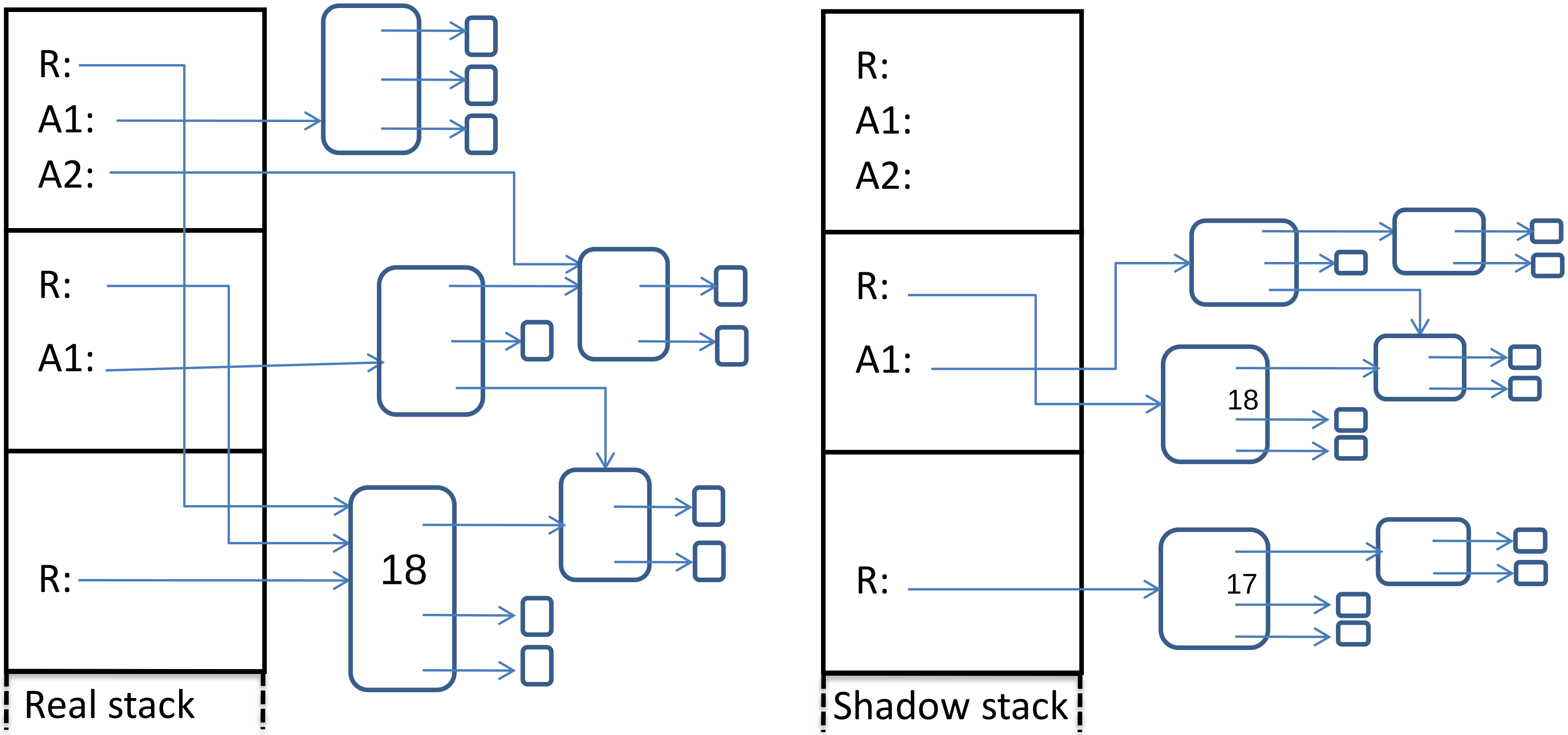
On method entry



Maintaining the shadow stack

On method entry:

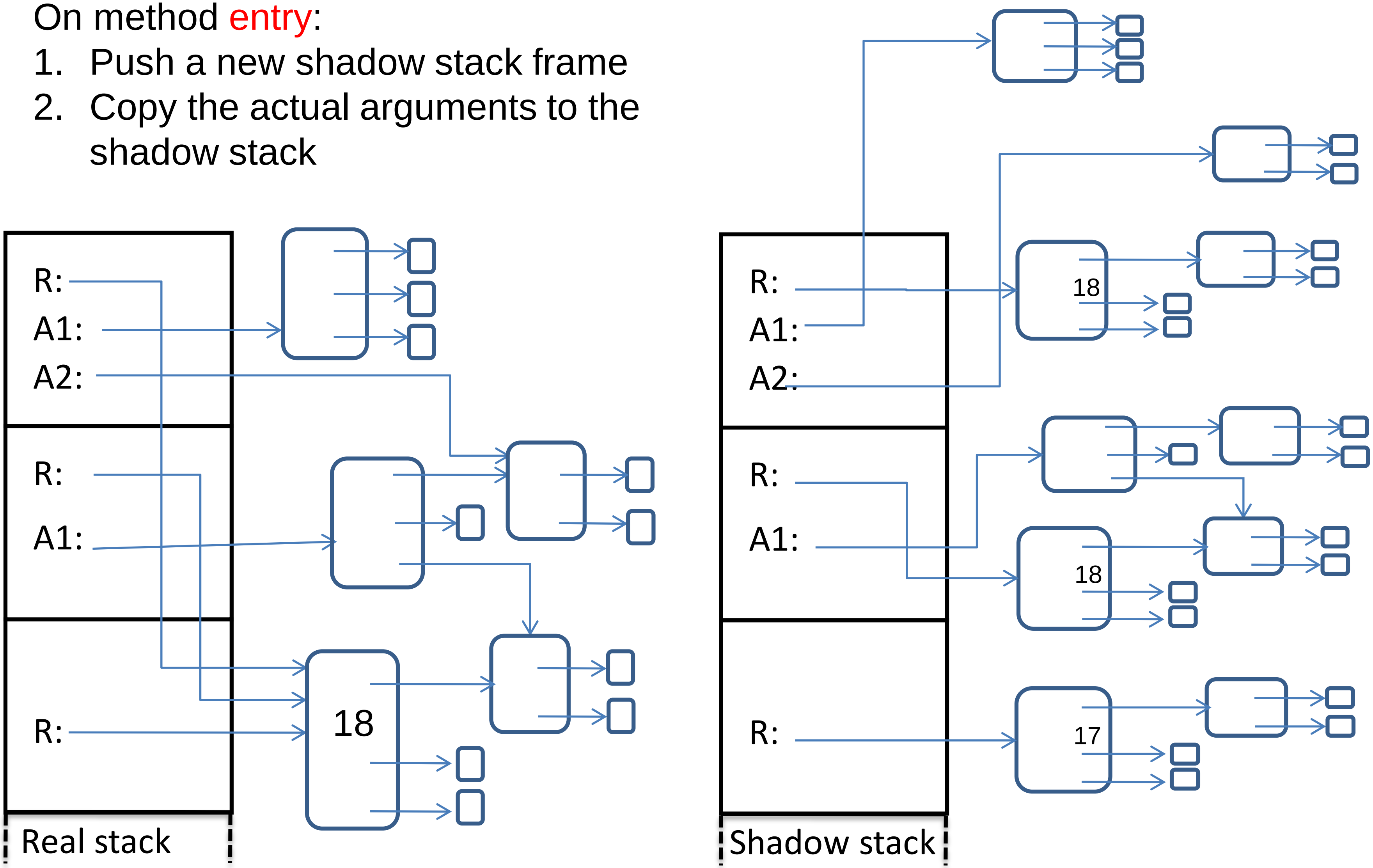
1. Push a new shadow stack frame



Maintaining the shadow stack

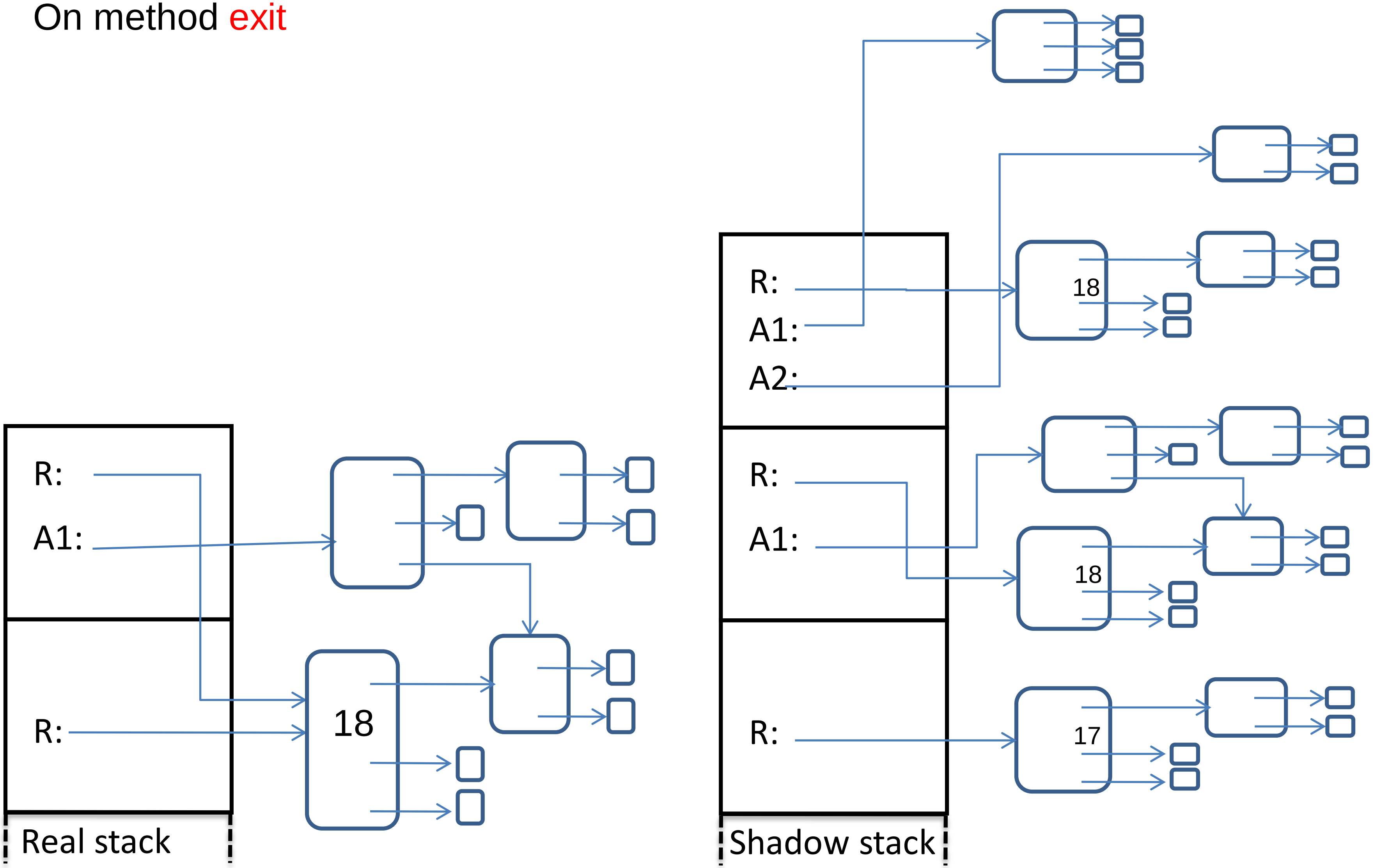
On method entry:

1. Push a new shadow stack frame
2. Copy the actual arguments to the shadow stack



Maintaining the shadow stack

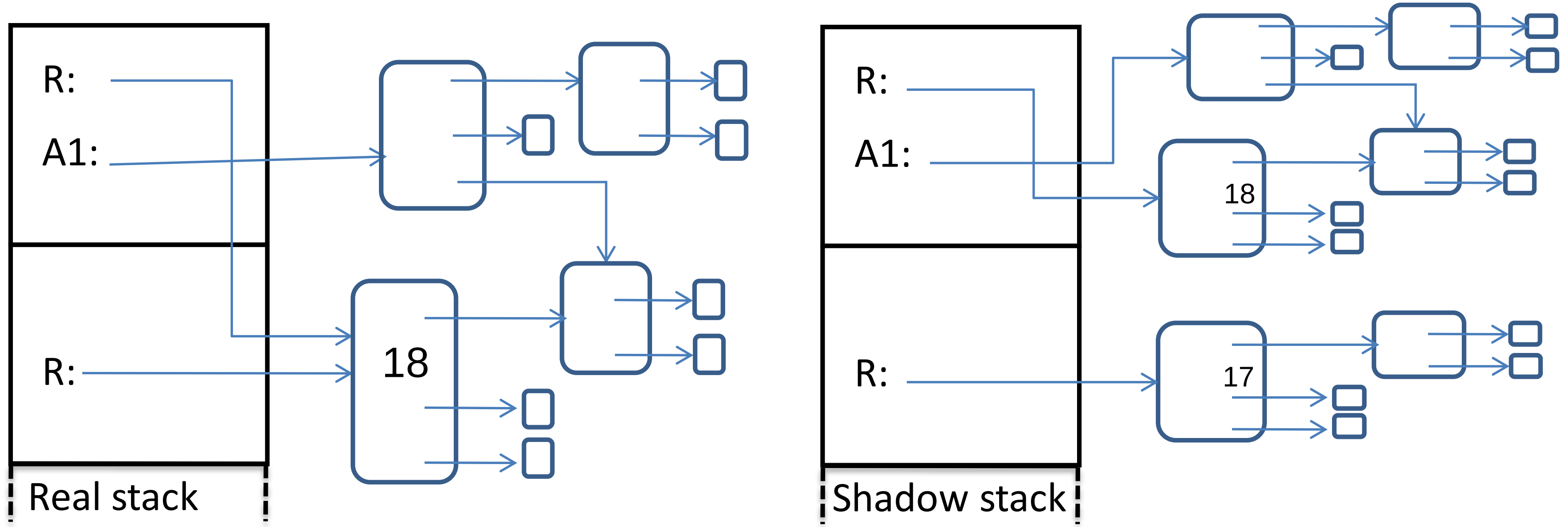
On method exit



Maintaining the shadow stack

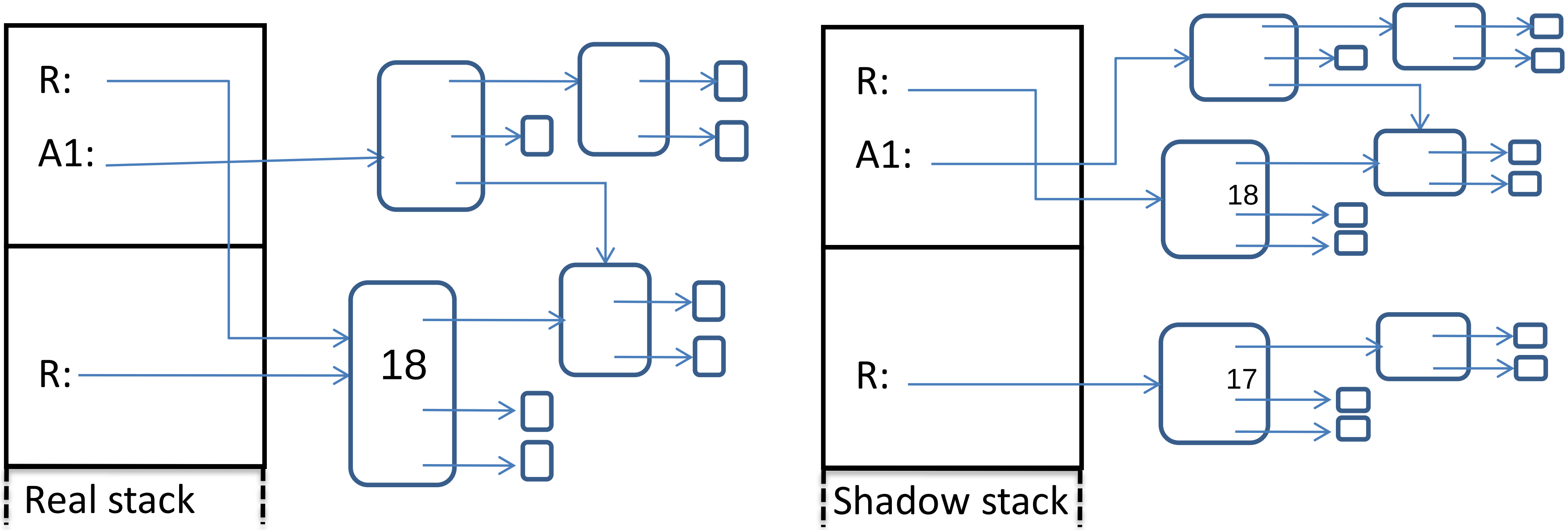
On method exit:

1. Pop shadow stack frame



Maintaining the shadow stack

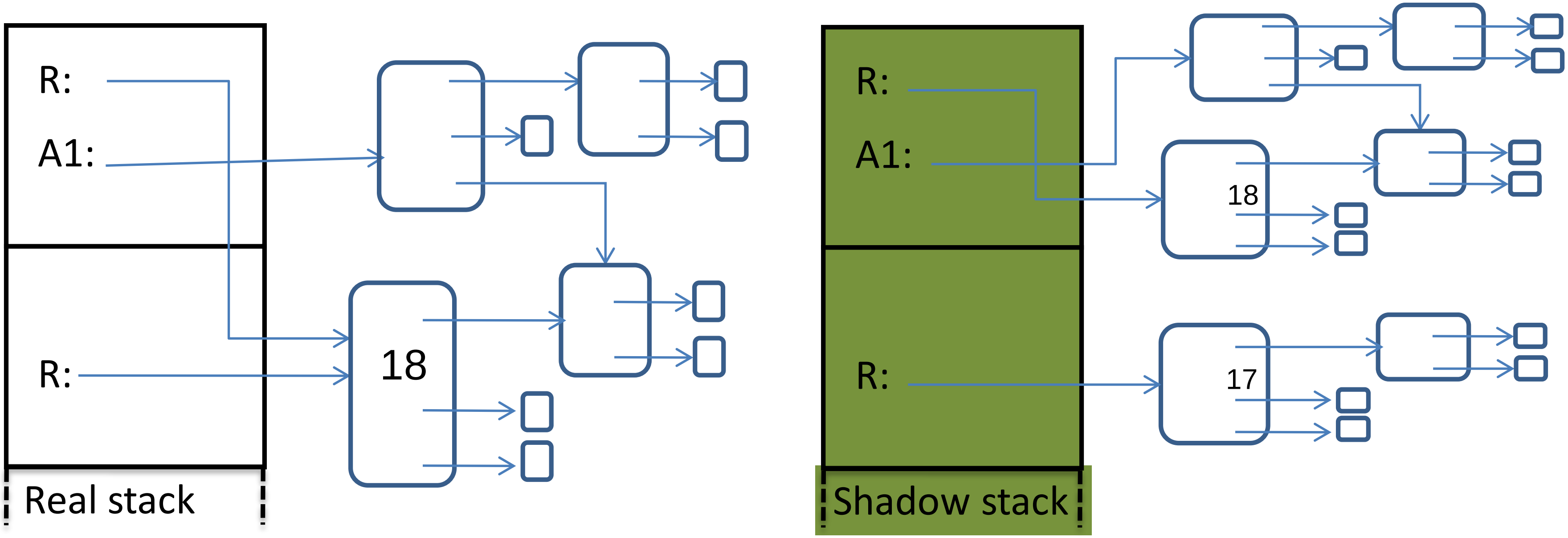
On program failure (top-level exception):



Maintaining the shadow stack

On program failure (top-level exception):

1. Write the shadow stack to a file

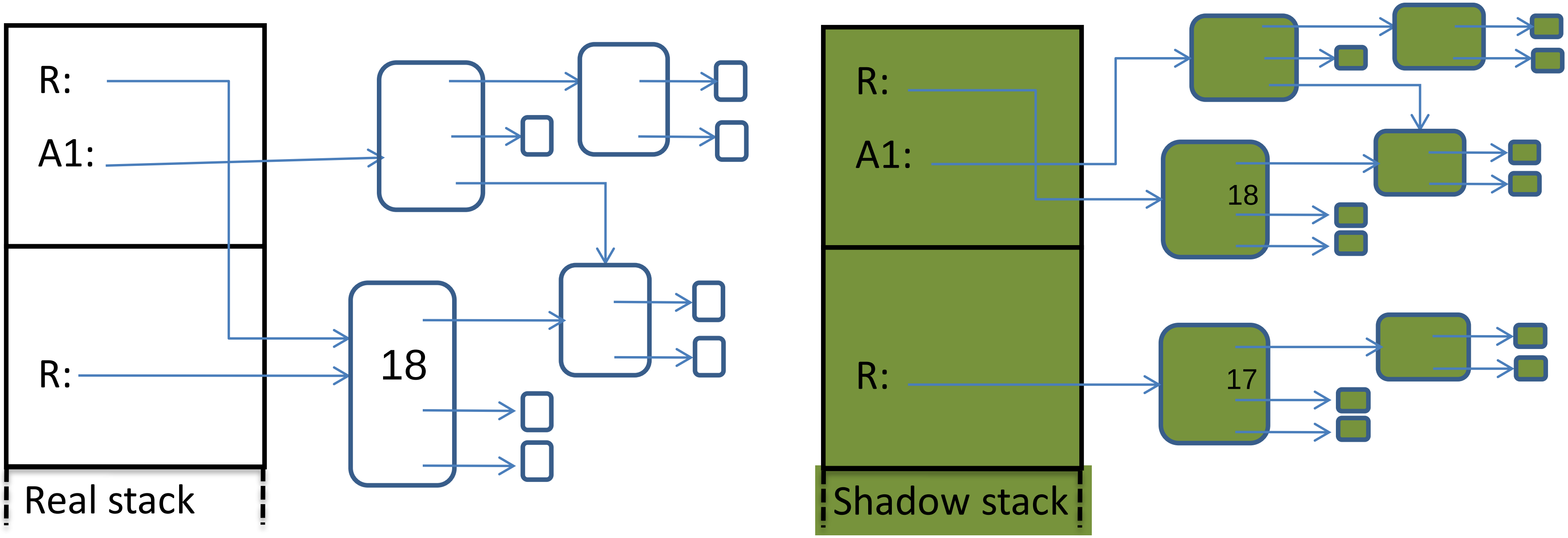


Maintaining the shadow stack

On program failure (top-level exception):

1. Write the shadow stack to a file

   Serializes all referenced state


R:
A1:
R:
18
Real stack
R:
A1:
R:
18
17
Shadow stack