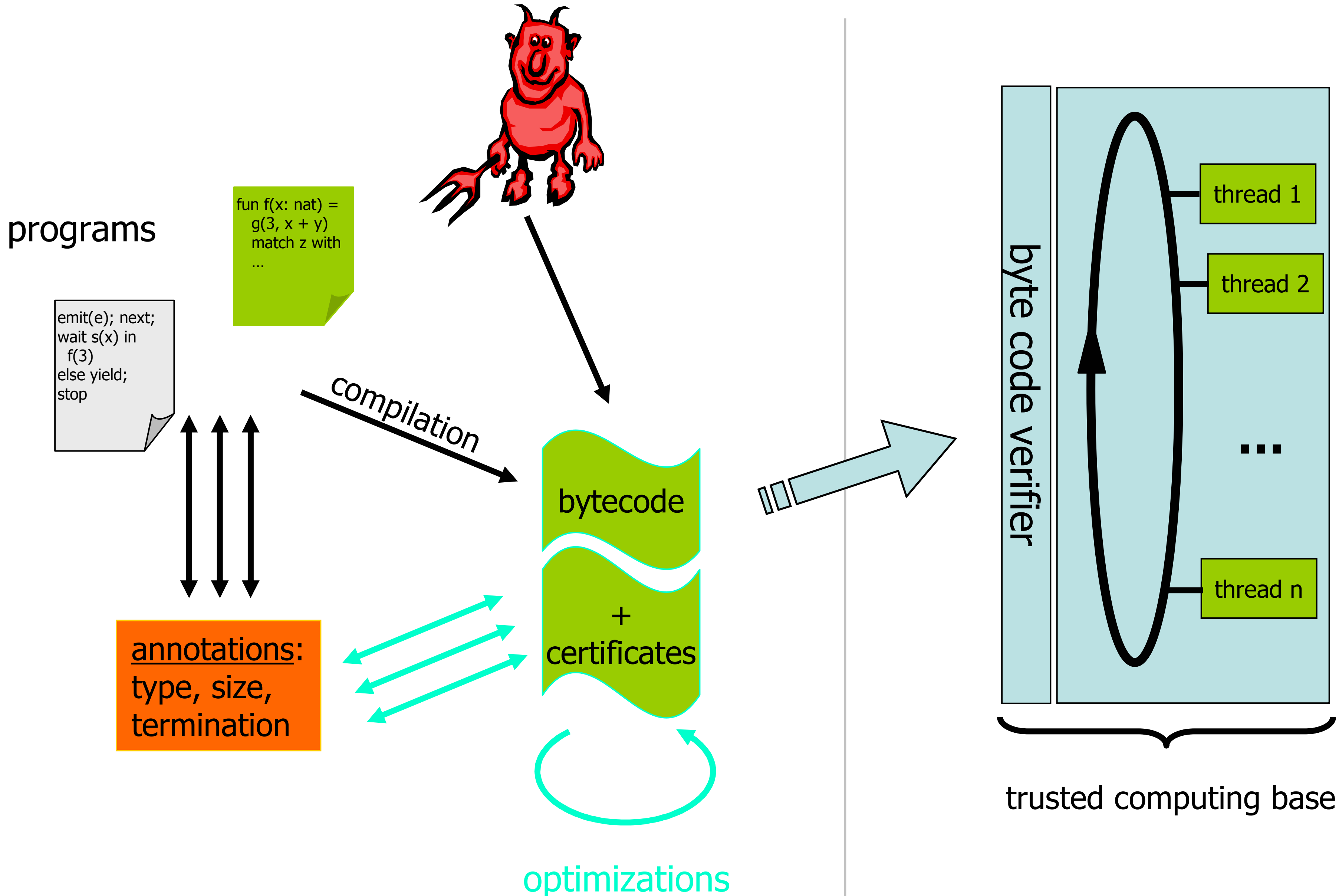
A Functional Scenario for Bytecode Verification of Space Bounds

Roberto Amadio, Solange Coupet-Grimal,
Silvano Dal Zilio and Line Jakubiec
LIF, Marseille (Fr)

Appeared as Research Report LIF 19-2004, January 2004


programs
fun f(x: nat) =
g(3, x + y)
match z with
...
emit(e); next;
wait s(x) in
f(3)
else yield;
stop
compilation
bytecode
+
certificates
annotations: type, size, termination
optimizations
byte code verifier
thread 1
thread 2
...
thread n
trusted computing base

A First-Order Functional Language

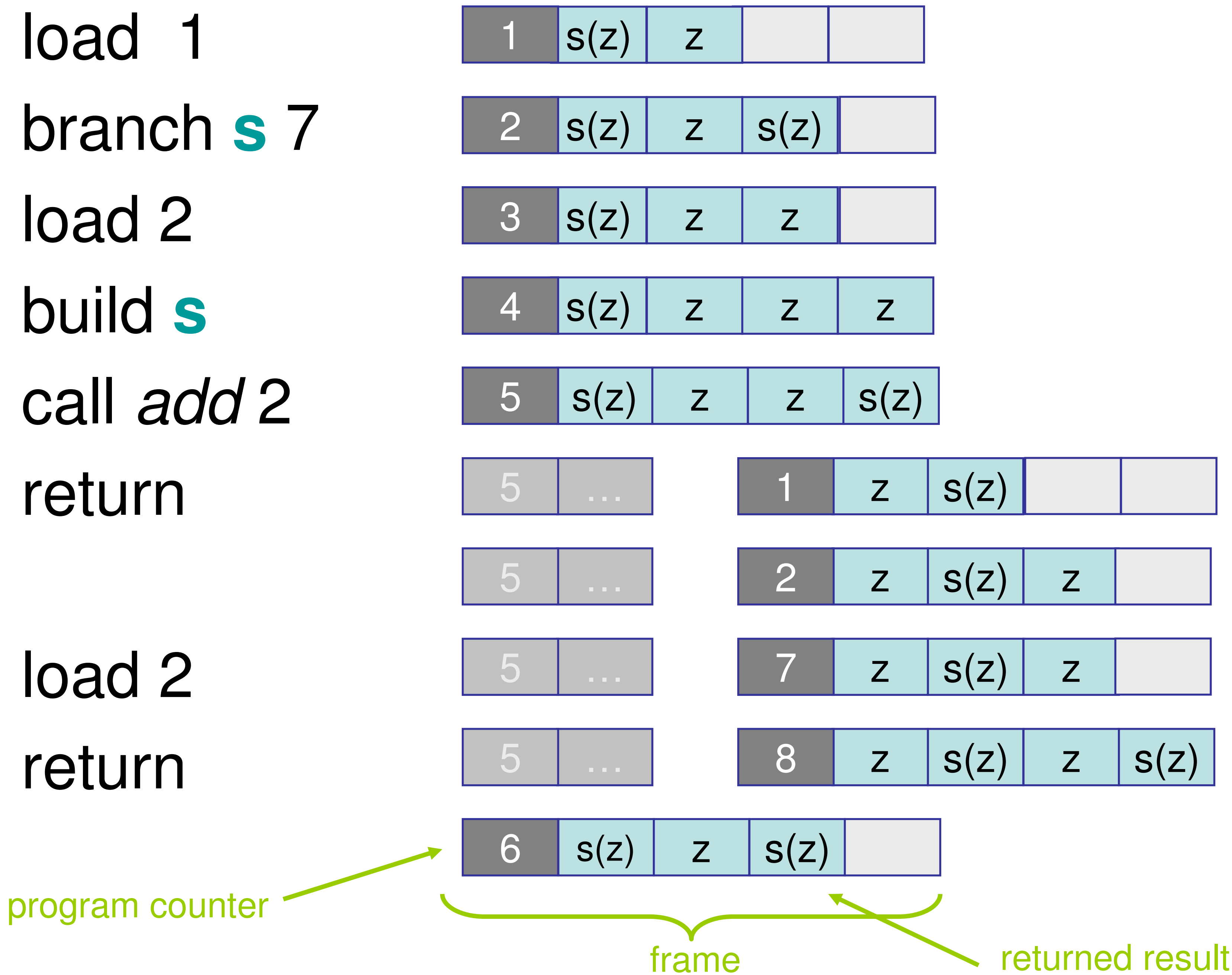
- With algebraic types, e.g. the type of natural numbers is nat ::= **z** | **s** *of* nat
- Functions and pattern-matching, e.g.
  add : (nat, nat) $\rightarrow$ nat :

$$
\begin{array}{lccl}
add & \mathsf{z} & y & = \; y \\
add & \mathsf{s}(x) & y & = \; add \; x \; \mathsf{s}(y)
\end{array}
$$

- Evaluation: add s(s(z)) s(z) $\Rightarrow$ s(s(s(z)))
- Compiled into bytecode instructions for a (simple) stack machine

1. load 1
2. branch **s** 7
3. load 2
4. build **s**
5. call *add* 2
6. return

7. load 2
8. return



Bounding the Space Needed

- It is easy to obtain a bound on the number of values in a frame $\rightarrow$ type verification
- We need to bound the size of the values $\rightarrow$ size verification, based on quasi-interpretations.
- We need to bound the number of frames in an execution path $\rightarrow$ termination verification, based on r.p.o. (recursive path orderings)

Bytecode (Type) Verification

1. load 1	nat nat
2. branch **s** 7	nat nat nat
3. load 2	nat nat nat
4. build **s**	nat nat nat nat
5. call *add* 2	nat nat nat nat
6. return	nat nat nat
7. load 2	nat nat nat
8. return	nat nat nat nat

Shape Verification

1.	load 1	x_1	x_2			
2.	branch **s** 7	x_1	x_2	x_1		
3.	load 2	$s(x_3)$	x_2	x_3		**$x_1 = s(x_3)$**
4.	build **s**	$s(x_3)$	x_2	x_3	x_2	
5.	call *add* 2	$s(x_3)$	x_2	x_3	$s(x_2)$	
6.	return	$s(x_3)$	x_2	$add(x_3, s(x_2))$		
7.	load 2	x_1	x_2	x_1		
8.	return	x_1	x_2	x_1	x_2	

Quasi-Interpretations

- The polynomial $q_{add}(x, y) = x + y$ is a valid quasi-interpretation for the function *add*.

- We can check that size information are correct (for “compiled programs”). In our example it amounts to check that:

$$q_{add}(1 + x_3, x_2) \geq q_{add}(x_3, 1 + x_2)$$

Termination

- We use termination criteria based on *recursive path ordering.*

- We can check that termination information are correct. In our example it amounts to check that:

$$\mathrm{add}(s(x_3), x_2) \quad >_l \quad \mathrm{add}(x_3, s(x_2))$$

Result

- A combination of polynomial quasi-interpretation and r.p.o. gives a (explicit!) polynomial upper-bound on the size needed for the execution.
- In our example, in an execution starting with the frame (add, 1, x_1 x_2):
  - a stack in a frame has size at most 4
  - every value has size less than $x_1 + x_2$
  - the number of frames is less than x_1

$$\text{size needed} \leq 4 \,.\, x_1 \,.\, (x_1 + x_2)$$

```
%%% type %%%
type nat = Z | S of nat
fun  nat add(nat,nat)


%%% size %%%
q_Z   = 0
q_S   = #1 + 1
q_add = #1 + #2


%%% termination %%%
exp, double > add
```

```
%%% code %%%
1: load 1
2: branch S 7
3: load 2
4: build S
5: call add 2
6: return
7: load 2
8: return
```