Seeing is Believing: Monitoring Future-Time Temporal Logic

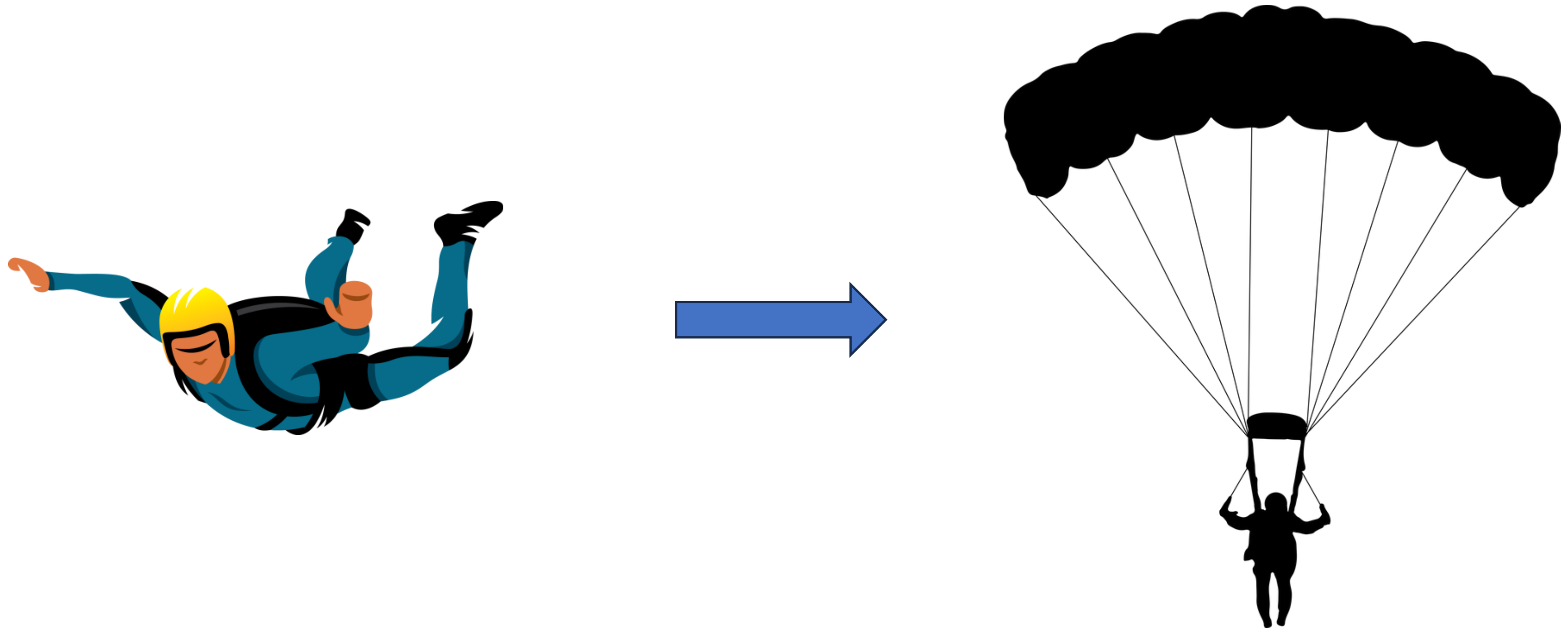
Max Fan

Summer 2023

Mentor: Ivan Perez Dominguez

A simplified example: skydiving

A simplified example: skydiving



The point of runtime monitoring

After jumping out the plane, the parachute will finally open within three seconds. → Trigger backup parachute

Globally, if the parachute is deployed, the person must decelerate. → Trigger backup parachute

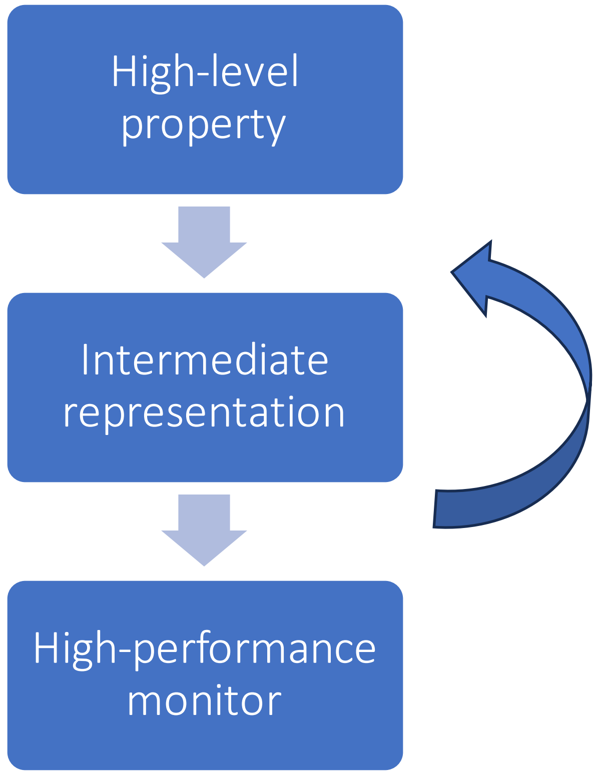
Globally, the person should not decelerate too fast. → Trigger call to 911

What would make this system robust?

You can jump out of a plane with it . . .

- Can reliably <u>detect</u> failures
- Can <u>handle</u> failures safely and immediately
- Can provide formal <u>guarantees</u> about error handling
- <u>Practical</u> for real-world use

Runtime monitoring is a PL problem



Copilot, LOLA, R2U2, etc.

Optimizations

C99, FPGA, etc.

Temporal Properties

Globally(x) = x is <u>globally</u> always true, in the future

Ex: G(if parachute_open() then decelerating())

Finally(x) = x will <u>finally</u> be true, in the future

Ex: F(parachute_open())

neXt(x) = x will be true at the <u>next</u> time step

Ex: X(parachute_deployed())

x Until y = x will be true <u>until</u> y becomes true

Ex: parachute_not_deployed() U parachute_is_released()

"The parachute will finally open"

F(parachute_open())

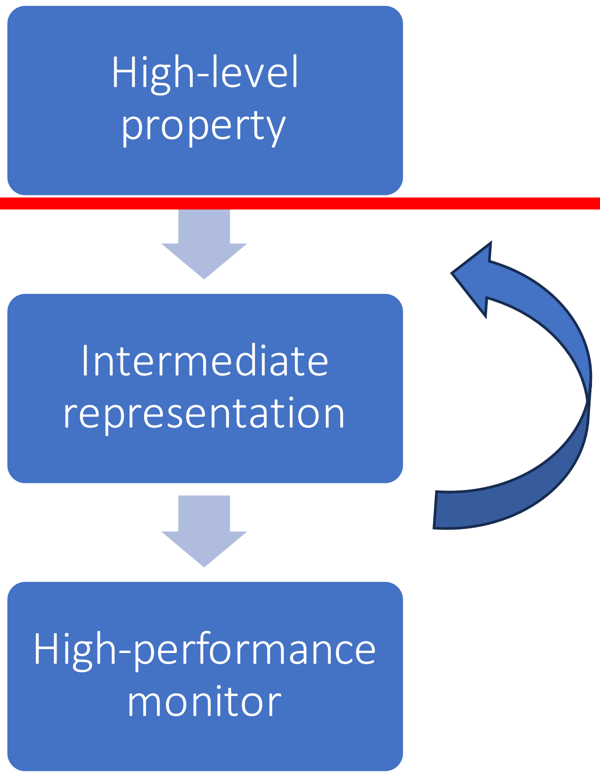
"The parachute will finally open in three seconds"

$F_{[0,3]}$(parachute_open())

Target

- Monitor systems online at runtime
- Support unbounded and bounded future-time formulae
- Actionable verdicts at every point
- Performant and practical

Monitor construction



A future-time unbounded temporal logic

A language that preserves the semantics

A language that is obviously monitorable

Intuition: "Seeing is Believing"

Consider: G(a)

Consider the trace:

0	1	2	3	4	5
a					

If we evaluate the formula, we get:

0	1	2	3	4	5
True so far					

Intuition: "Seeing is Believing"

Consider: G(a)

Consider the trace:

0	1	2	3	4	5
a	a				

If we evaluate the formula, we get:

0	1	2	3	4	5
True so far	True so far				

Intuition: "Seeing is Believing"

Consider: G(a)

Consider the trace:

0	**1**	**2**	**3**	**4**	**5**
a	a	a	a	a	a

If we evaluate the formula, we get:

0	**1**	**2**	**3**	**4**	**5**
True so far	True so far	True so far	True so far	True so far	True so far

Intuition: "Seeing is Believing"

Consider: G(a)

Consider the trace:

0	1	2	3	4	5
a	a	not a	a	a	a

If we evaluate the formula, we get:

0	1	2	3	4	5
True so far	True so far	False	False	False	False

Semantics overview

True	= we have seen the evidence that the formula is true
True so far	= if the stream stops now, the formula is true
False so far	= if the stream stops now, the formula is false
False	= we have seen the evidence that the formula is false

Making things more formal

For n <= m, we can recursively define the evaluation of a formula at n with trace length m+1:

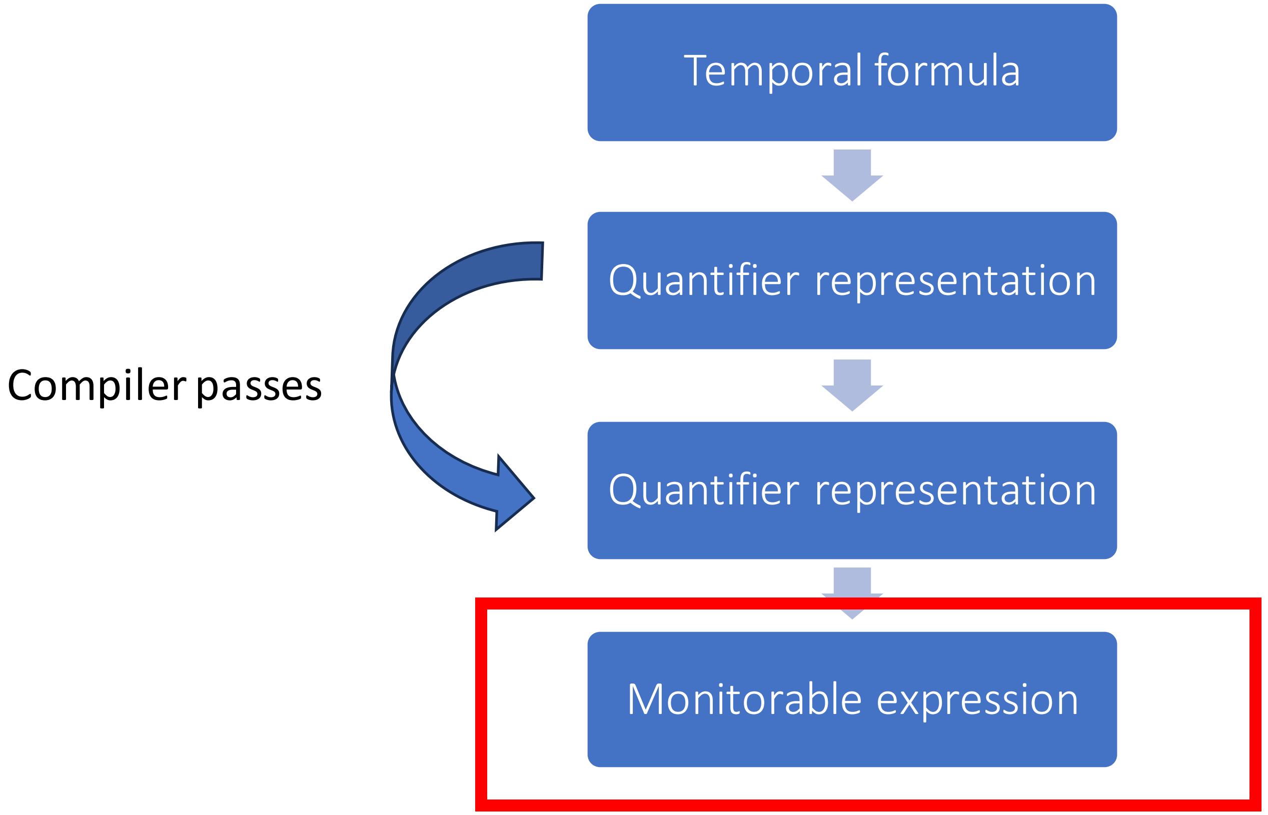
$$\boxed{[\![G(\phi)]\!]_{n}^{m+1} = \forall i \in [n...m], [\![\phi]\!]_{i}^{m+1}}$$

$$[\![F(\phi)]\!]_{n}^{m+1} = \exists i \in [n...m], [\![\phi]\!]_{i}^{m+1}$$

$$[\![\phi U \psi]\!]_{n}^{m+1} = \exists i \in [n...m], (\forall j \in [n...(i-1)], [\![\phi]\!]_{j}^{m+1}) \wedge [\![\psi]\!]_{i}^{m+1}$$

$$[\![X(\phi)]\!]_{n}^{m+1} = \exists i \in [n+1], [\![\phi]\!]_{i}^{m+1}$$

Monitor construction

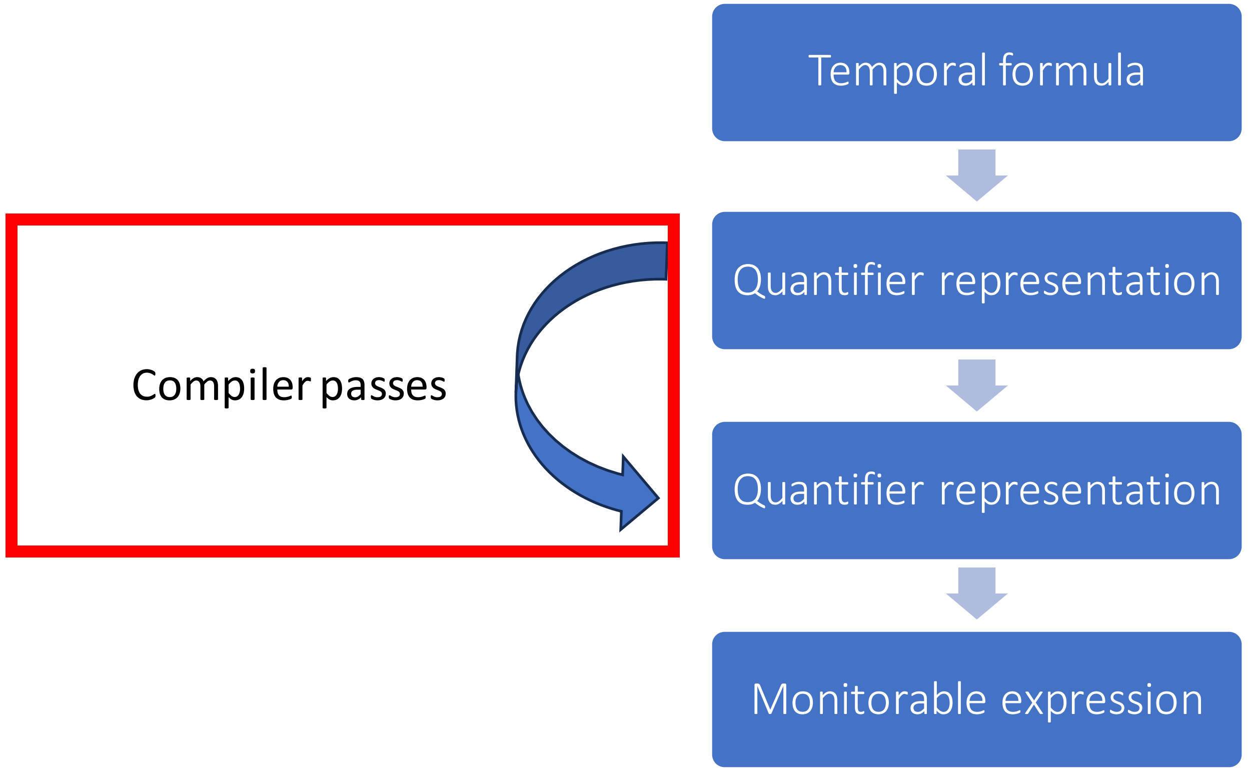


What is a monitorable expression?

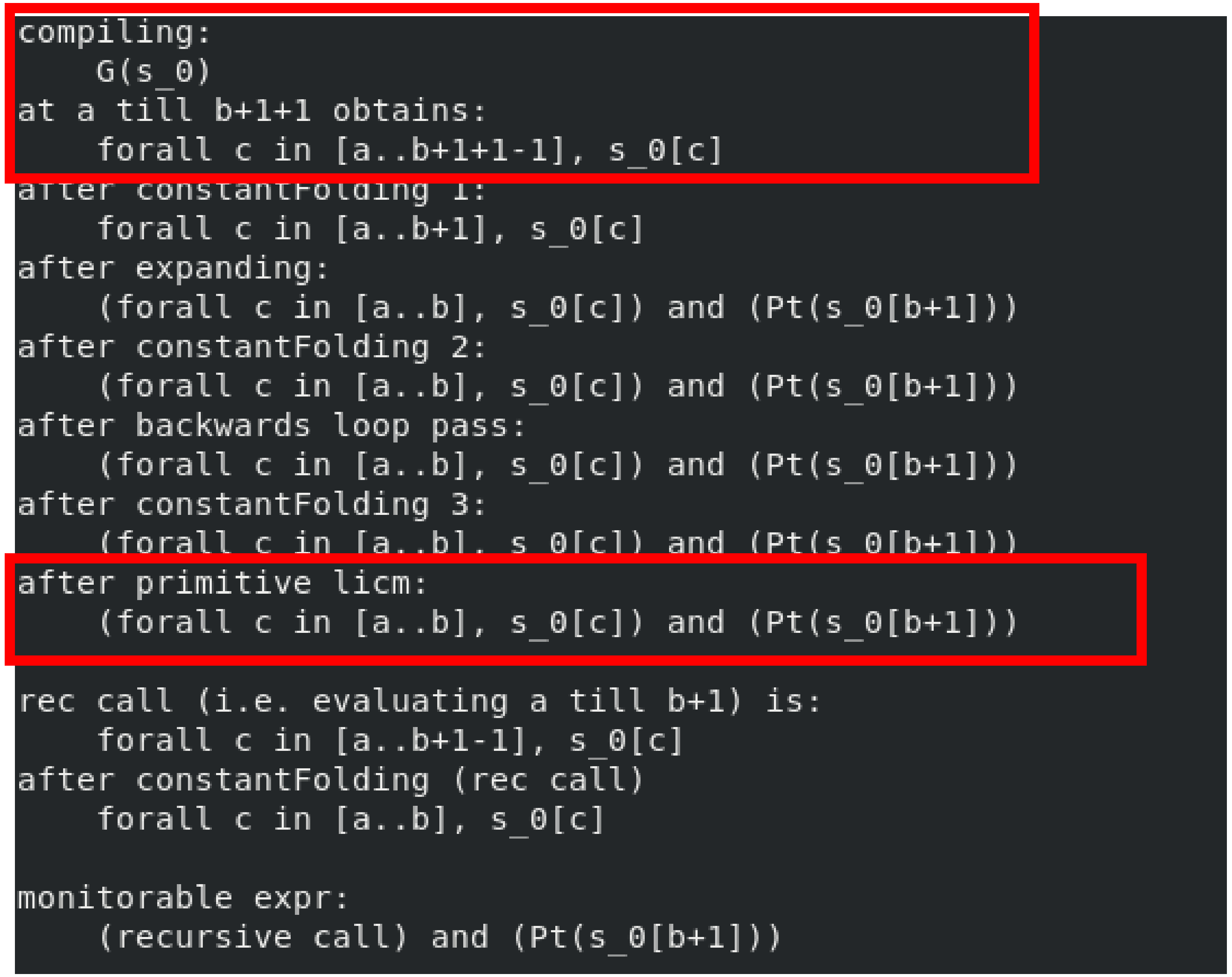
The form of a monitorable expression:

"current verdict := f(previous verdict, current values)"

Monitor construction



Example 1

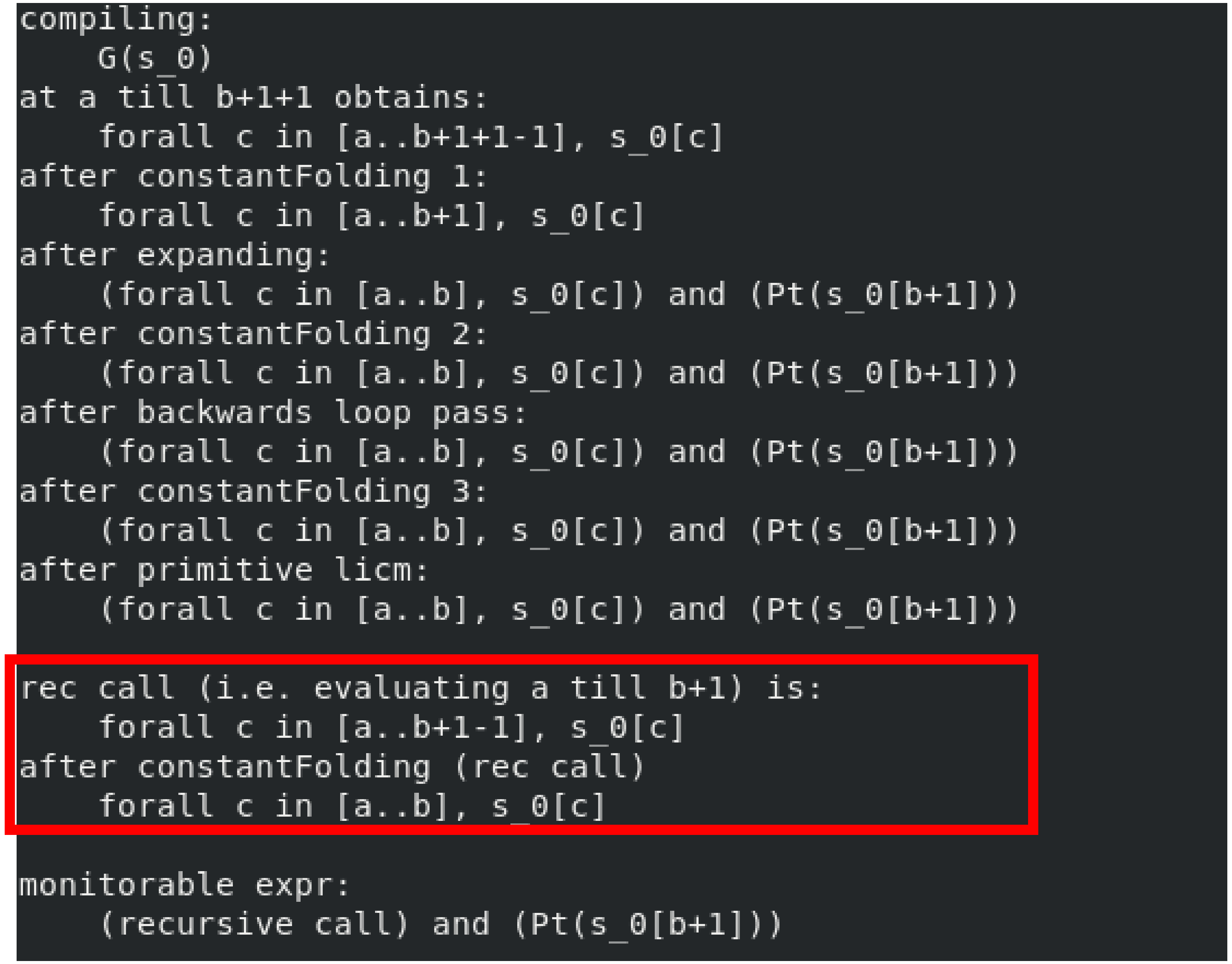


```
compiling:
    G(s_0)
at a till b+1+1 obtains:
    forall c in [a..b+1+1-1], s_0[c]
after constantFolding 1:
    forall c in [a..b+1], s_0[c]
after expanding:
    (forall c in [a..b], s_0[c]) and (Pt(s_0[b+1]))
after constantFolding 2:
    (forall c in [a..b], s_0[c]) and (Pt(s_0[b+1]))
after backwards loop pass:
    (forall c in [a..b], s_0[c]) and (Pt(s_0[b+1]))
after constantFolding 3:
    (forall c in [a..b], s_0[c]) and (Pt(s_0[b+1]))
after primitive licm:
    (forall c in [a..b], s_0[c]) and (Pt(s_0[b+1]))

rec call (i.e. evaluating a till b+1) is:
    forall c in [a..b+1-1], s_0[c]
after constantFolding (rec call)
    forall c in [a..b], s_0[c]

monitorable expr:
    (recursive call) and (Pt(s_0[b+1]))
```

Example 1

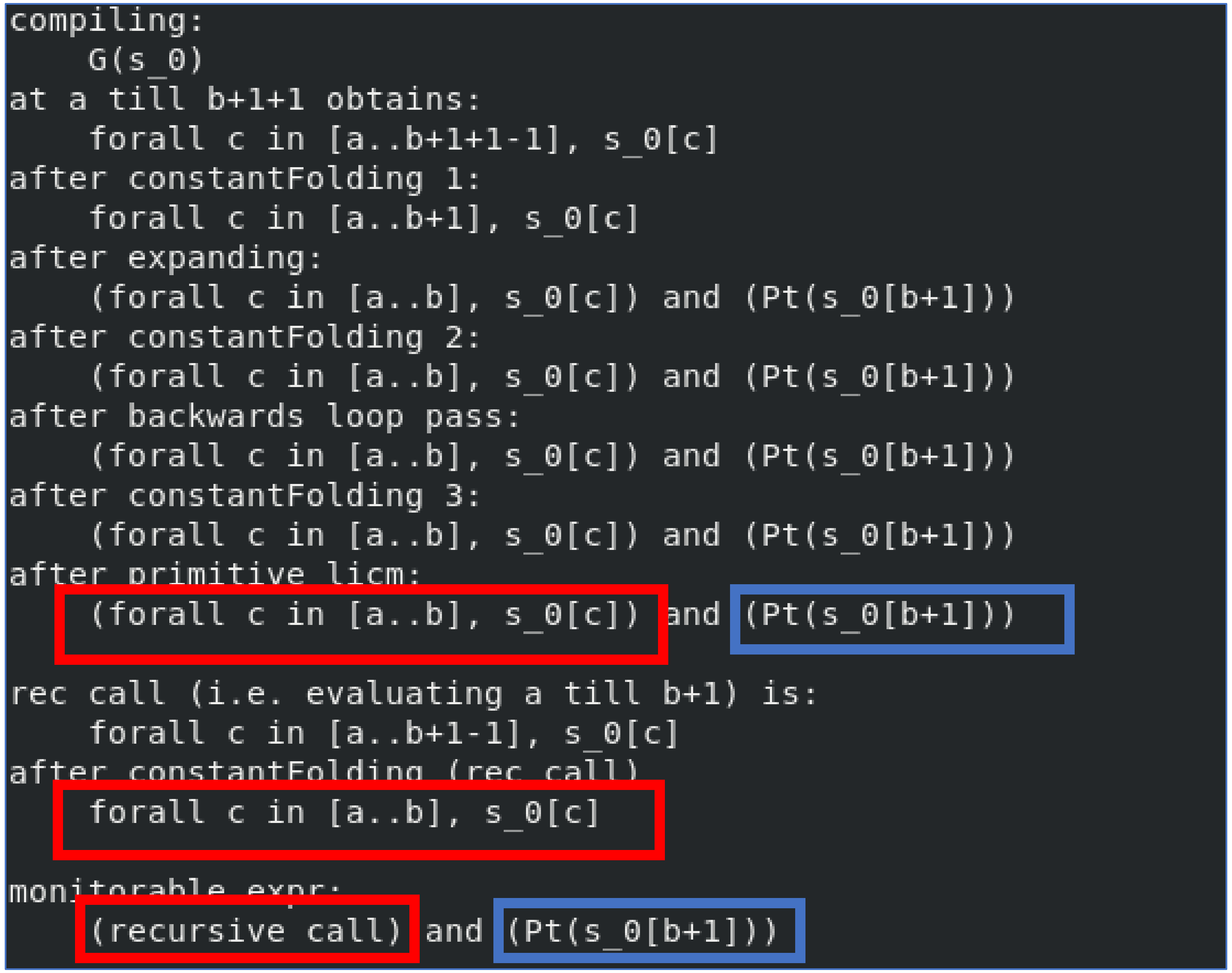


```
compiling:
    G(s_0)
at a till b+1+1 obtains:
    forall c in [a..b+1+1-1], s_0[c]
after constantFolding 1:
    forall c in [a..b+1], s_0[c]
after expanding:
    (forall c in [a..b], s_0[c]) and (Pt(s_0[b+1]))
after constantFolding 2:
    (forall c in [a..b], s_0[c]) and (Pt(s_0[b+1]))
after backwards loop pass:
    (forall c in [a..b], s_0[c]) and (Pt(s_0[b+1]))
after constantFolding 3:
    (forall c in [a..b], s_0[c]) and (Pt(s_0[b+1]))
after primitive licm:
    (forall c in [a..b], s_0[c]) and (Pt(s_0[b+1]))

rec call (i.e. evaluating a till b+1) is:
    forall c in [a..b+1-1], s_0[c]
after constantFolding (rec call)
    forall c in [a..b], s_0[c]

monitorable expr:
    (recursive call) and (Pt(s_0[b+1]))
```

Example 1

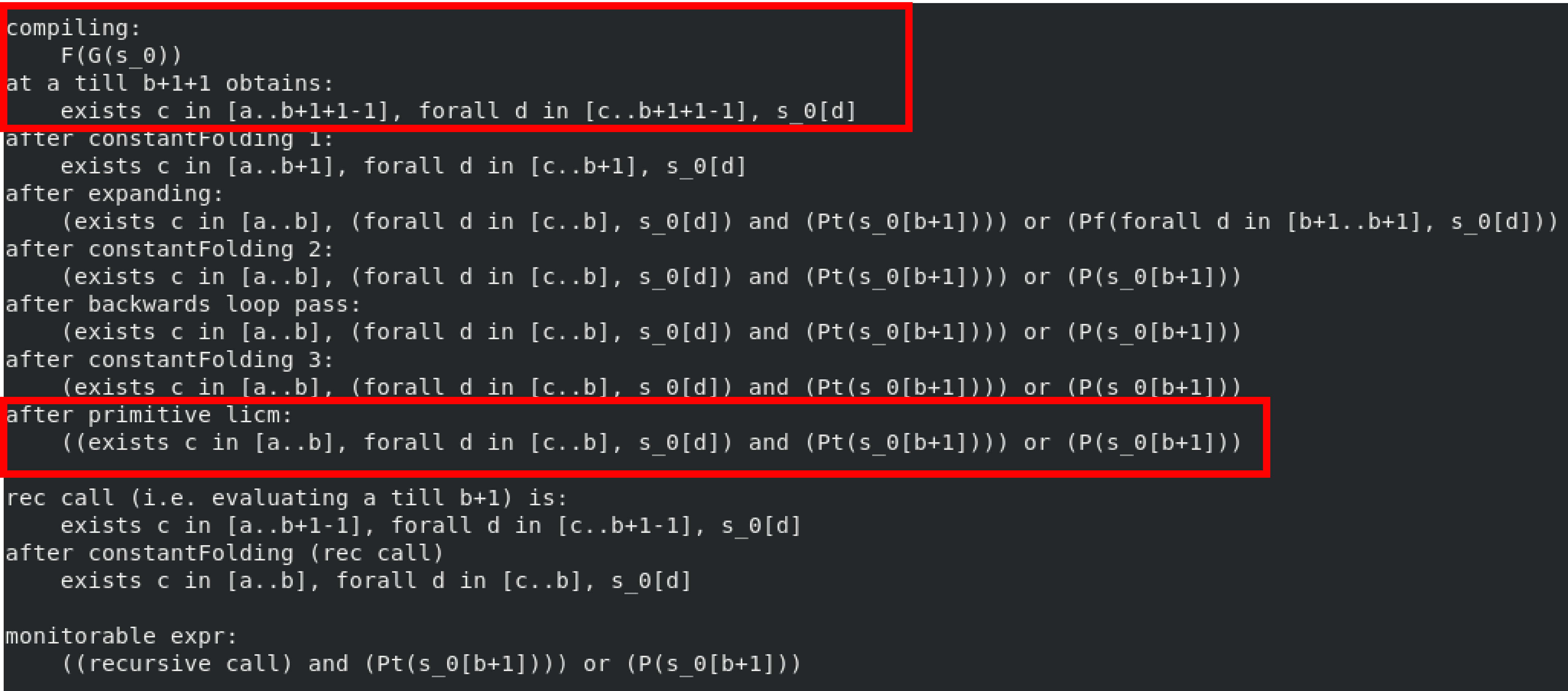


```
compiling:
    G(s_0)
at a till b+1+1 obtains:
    forall c in [a..b+1+1-1], s_0[c]
after constantFolding 1:
    forall c in [a..b+1], s_0[c]
after expanding:
    (forall c in [a..b], s_0[c]) and (Pt(s_0[b+1]))
after constantFolding 2:
    (forall c in [a..b], s_0[c]) and (Pt(s_0[b+1]))
after backwards loop pass:
    (forall c in [a..b], s_0[c]) and (Pt(s_0[b+1]))
after constantFolding 3:
    (forall c in [a..b], s_0[c]) and (Pt(s_0[b+1]))
after primitive licm:
    (forall c in [a..b], s_0[c]) and (Pt(s_0[b+1]))

rec call (i.e. evaluating a till b+1) is:
    forall c in [a..b+1-1], s_0[c]
after constantFolding (rec call)
    forall c in [a..b], s_0[c]

monitorable expr:
    (recursive call) and (Pt(s_0[b+1]))
```

Example 2

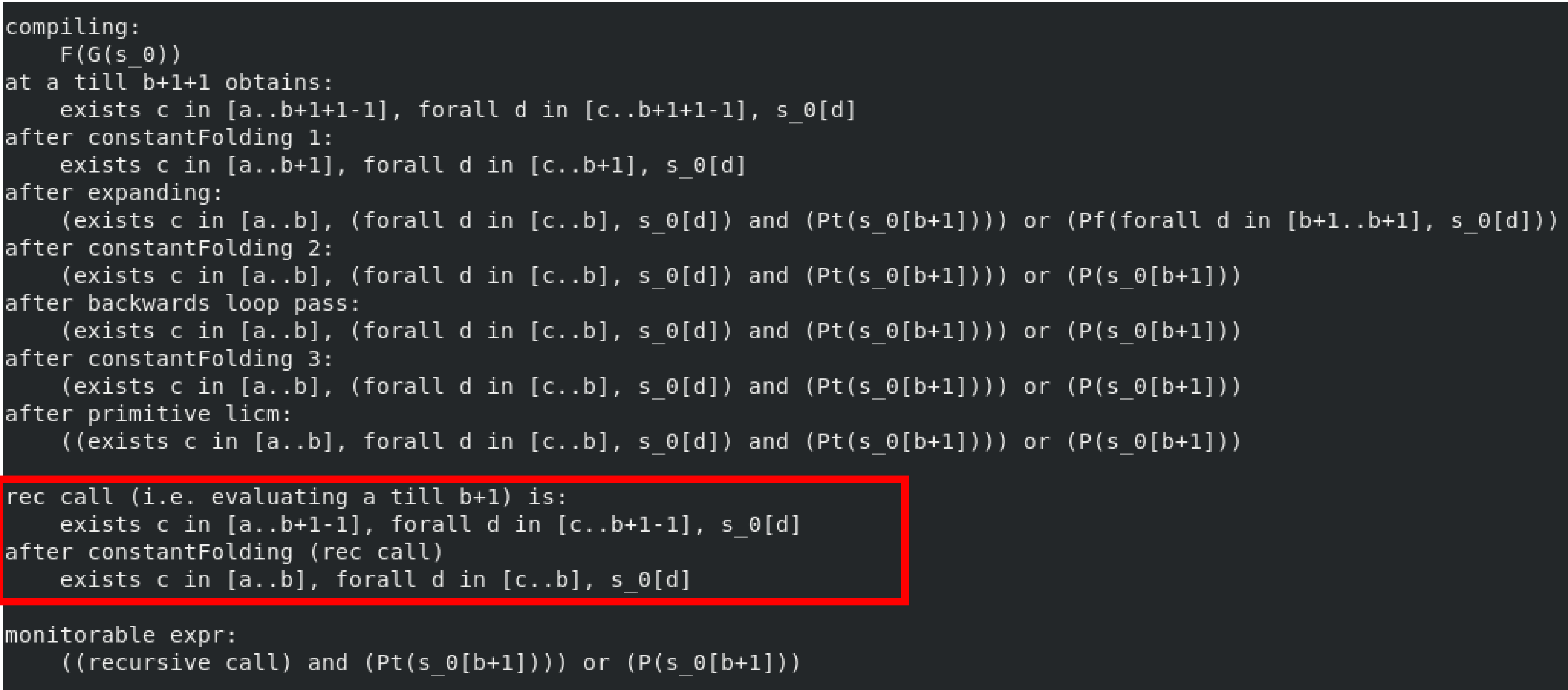


```
compiling:
    F(G(s_0))
at a till b+1+1 obtains:
    exists c in [a..b+1+1-1], forall d in [c..b+1+1-1], s_0[d]
after constantFolding 1:
    exists c in [a..b+1], forall d in [c..b+1], s_0[d]
after expanding:
    (exists c in [a..b], (forall d in [c..b], s_0[d]) and (Pt(s_0[b+1]))) or (Pf(forall d in [b+1..b+1], s_0[d]))
after constantFolding 2:
    (exists c in [a..b], (forall d in [c..b], s_0[d]) and (Pt(s_0[b+1]))) or (P(s_0[b+1]))
after backwards loop pass:
    (exists c in [a..b], (forall d in [c..b], s_0[d]) and (Pt(s_0[b+1]))) or (P(s_0[b+1]))
after constantFolding 3:
    (exists c in [a..b], (forall d in [c..b], s_0[d]) and (Pt(s_0[b+1]))) or (P(s_0[b+1]))
after primitive licm:
    ((exists c in [a..b], forall d in [c..b], s_0[d]) and (Pt(s_0[b+1]))) or (P(s_0[b+1]))

rec call (i.e. evaluating a till b+1) is:
    exists c in [a..b+1-1], forall d in [c..b+1-1], s_0[d]
after constantFolding (rec call)
    exists c in [a..b], forall d in [c..b], s_0[d]

monitorable expr:
    ((recursive call) and (Pt(s_0[b+1]))) or (P(s_0[b+1]))
```

Example 2

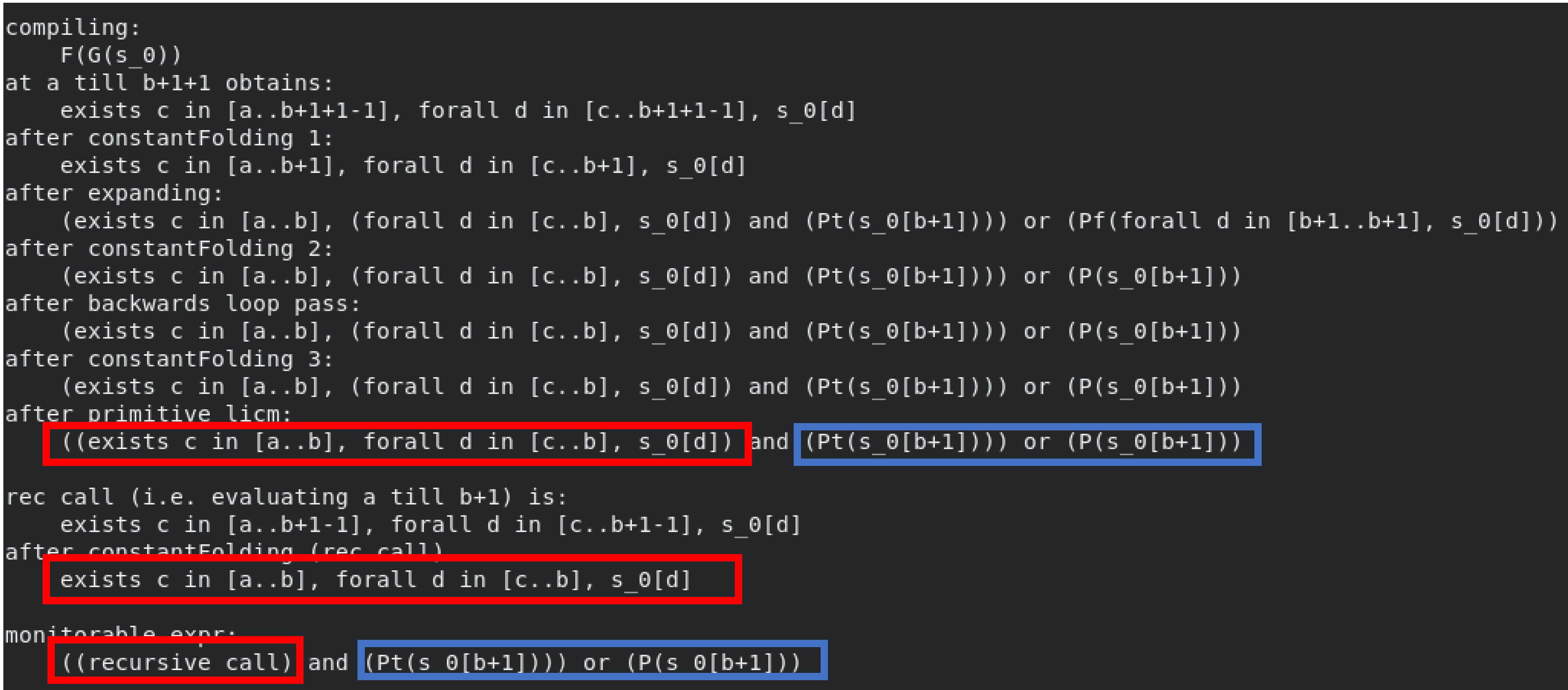


```
compiling:
    F(G(s_0))
at a till b+1+1 obtains:
    exists c in [a..b+1+1-1], forall d in [c..b+1+1-1], s_0[d]
after constantFolding 1:
    exists c in [a..b+1], forall d in [c..b+1], s_0[d]
after expanding:
    (exists c in [a..b], (forall d in [c..b], s_0[d]) and (Pt(s_0[b+1]))) or (Pf(forall d in [b+1..b+1], s_0[d]))
after constantFolding 2:
    (exists c in [a..b], (forall d in [c..b], s_0[d]) and (Pt(s_0[b+1]))) or (P(s_0[b+1]))
after backwards loop pass:
    (exists c in [a..b], (forall d in [c..b], s_0[d]) and (Pt(s_0[b+1]))) or (P(s_0[b+1]))
after constantFolding 3:
    (exists c in [a..b], (forall d in [c..b], s_0[d]) and (Pt(s_0[b+1]))) or (P(s_0[b+1]))
after primitive licm:
    ((exists c in [a..b], forall d in [c..b], s_0[d]) and (Pt(s_0[b+1]))) or (P(s_0[b+1]))

rec call (i.e. evaluating a till b+1) is:
    exists c in [a..b+1-1], forall d in [c..b+1-1], s_0[d]
after constantFolding (rec call)
    exists c in [a..b], forall d in [c..b], s_0[d]

monitorable expr:
    ((recursive call) and (Pt(s_0[b+1]))) or (P(s_0[b+1]))
```

Example 2



```
compiling:
    F(G(s_0))
at a till b+1+1 obtains:
    exists c in [a..b+1+1-1], forall d in [c..b+1+1-1], s_0[d]
after constantFolding 1:
    exists c in [a..b+1], forall d in [c..b+1], s_0[d]
after expanding:
    (exists c in [a..b], (forall d in [c..b], s_0[d]) and (Pt(s_0[b+1]))) or (Pf(forall d in [b+1..b+1], s_0[d]))
after constantFolding 2:
    (exists c in [a..b], (forall d in [c..b], s_0[d]) and (Pt(s_0[b+1]))) or (P(s_0[b+1]))
after backwards loop pass:
    (exists c in [a..b], (forall d in [c..b], s_0[d]) and (Pt(s_0[b+1]))) or (P(s_0[b+1]))
after constantFolding 3:
    (exists c in [a..b], (forall d in [c..b], s_0[d]) and (Pt(s_0[b+1]))) or (P(s_0[b+1]))
after primitive licm:
    ((exists c in [a..b], forall d in [c..b], s_0[d]) and (Pt(s_0[b+1]))) or (P(s_0[b+1]))

rec call (i.e. evaluating a till b+1) is:
    exists c in [a..b+1-1], forall d in [c..b+1-1], s_0[d]
after constantFolding (rec call)
    exists c in [a..b], forall d in [c..b], s_0[d]

monitorable expr:
    ((recursive call) and (Pt(s_0[b+1]))) or (P(s_0[b+1]))
```

Other approaches in the literature

- RVLTL (theoretical)
  - **Expressivity:** Unbounded future-time
  - **Verdicts:** Four-valued verdict, online
  - **Performance:** Monitors take double-exponential space
- R2U2 (applied)
  - **Expressivity:** Only bounded future-time
  - **Verdict:** True/false verdict, with delays
  - **Performance:** Monitors are efficient

Primary Contributions

It is possible to:

- Use a future-time, unbounded temporal logic
  (previously with R2U2, only bounded)

- Monitor in poly(?) space and time in size of the property
  (previously with RVLTL, 2-EXP space, NP-hard and PSPACE-hard)

- Produce an actionable verdict at every point in time
  (previously with R2U2, verdicts can be delayed)

Comparison

	Unbounded temporal operators	Worst-case Monitor Space Complexity*	Always produces actionable verdict
RVLTL	✓	O(2^2^N)	✓
R2U2	✗	O(N*M)?	✗
Mine	✓	POLY?	✓

* Space complexity is denoted in size of the formula, N. For R2U2, M denotes the maximum time bound in the formula. All three runtime monitoring systems are constant in size of the input stream).

Intuition: RVLTL

Consider: X((a and b and c) or (a and (not b)) or …)

RVLTL's semantics <u>demand</u> that the monitor perform LTL satisfiability checking.

The monitor returns True iff the formula is TAUT.

The monitor returns False iff the formula is UNSAT.

The monitor returns other values iff the formula is SAT.

Intuition: "Seeing is Believing"

Consider: X((a and b and c) or (a and (not b)) or …)

Consider the trace:

0	1	2	3	4	5
a, b, not c	a, b, c	a, b, c	a, b, c	a, b, c	a, b, c

If we evaluate the formula, we get:

$\boxed{0}$	1	2	3	4	5
$\boxed{?}$	True	True	True	True	True

Intuition: "Seeing is Believing"

Consider: X((a and b and c) or (a and (not b)) or …)

Consider the trace:

0	1	2	3	4	5
a, b, not c	a, b, c	a, b, c	a, b, c	a, b, c	a, b, c

If we evaluate the formula, we get:

0	1	2	3	4	5
False so far	True	True	True	True	True

Soundness for temporal logic on infinite streams

True	=> Formula is True on infinite continuations of the stream
True so far	=> N/A
False so far	=> N/A
False	=> Formula is False on infinite continuations of the stream

Seeing is Believing: A More Explainable Semantics for Future-Time Unbounded Temporal Logic in Runtime Monitoring

Max Fan
NASA
maxwell.y.fan@nasa.gov

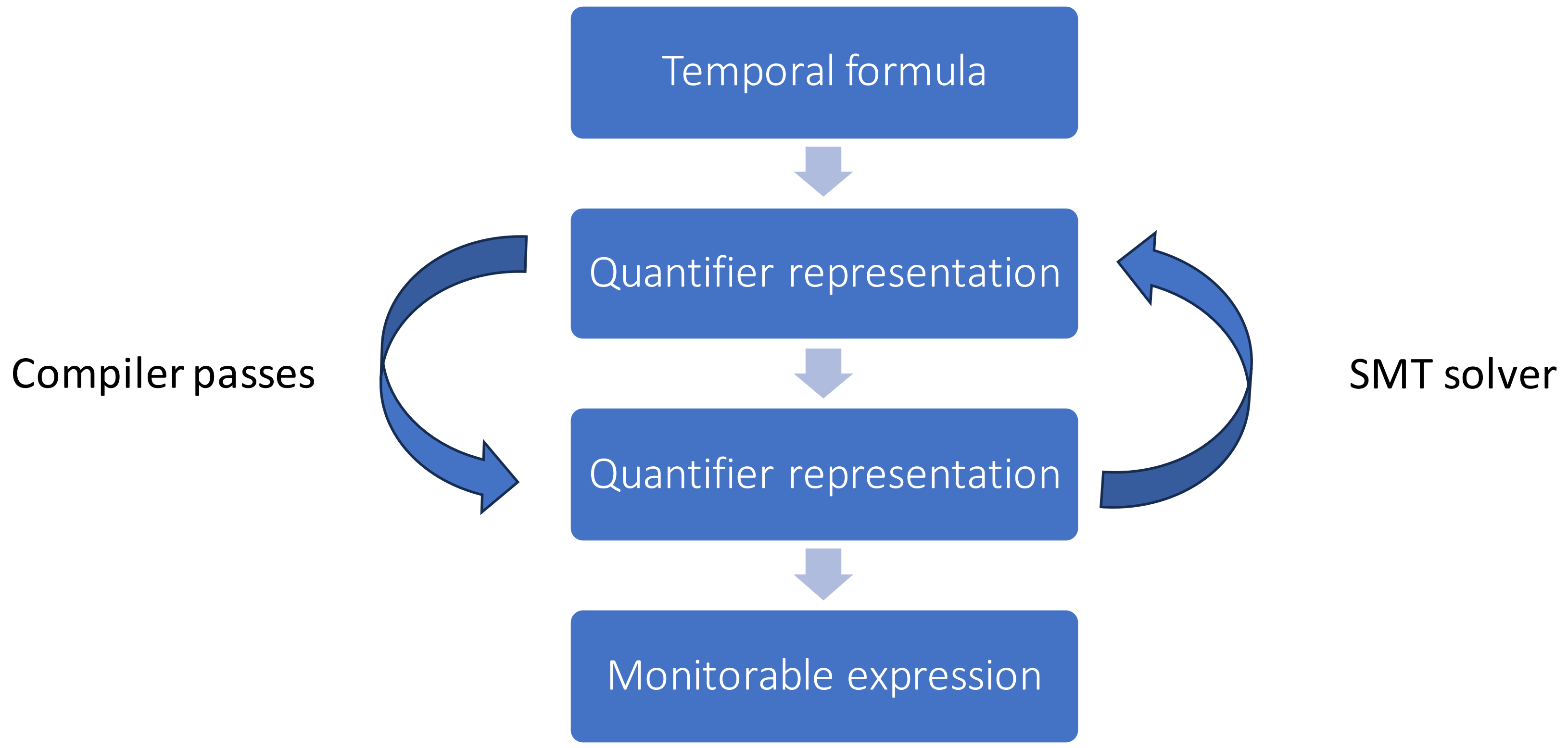
Ivan Perez
KBR @ NASA Ames Research Center
ivan.perezdominguez@nasa.gov

August 2023

Abstract

Runtime monitors for future-time unbounded temporal logics like RVLTL,

Idea: Verifying the compiler



Note: This is NOT implemented yet. Just an idea . . .

Ideas for Future Work

- Finish the monitoring algorithm in Copilot
- Explore connection with polyhedral compilation and geometry
- Mechanize correctness and soundness proofs, perhaps in Coq
- Evaluate real-world performance against competitors

What else was I up to this summer?

- Discovered and fixed a soundness bug in the Copilot compiler
- Extended the Copilot interpreter and compiler to be able to modify streams of arrays in-place
- Discovered and wrote patches for performance issues in the Copilot interpreter
- Implemented a future-time bounded temporal logic
- Wrote monitors for UAV flights
- Wrote cute, miscellaneous proofs about Copilot
- Developed a future-time unbounded temporal logic that improves on prior work in the literature, paper in progress

Thanks

Thanks to Ivan, Tom, and Esther for supporting me on this project.

Thanks to Gricel, Karan, Luisa, Steven, Beverly, Jonathan, Mari, Elle, Aiden, Rachel, and Morgan for being awesome interns!

Extra slides!

Idea: Compiling For Unreliable Hardware

Modern compilers assume reliable hardware.

This is not suitable for space.

Can we bootstrap **reliable** software from ***unreliable*** hardware?

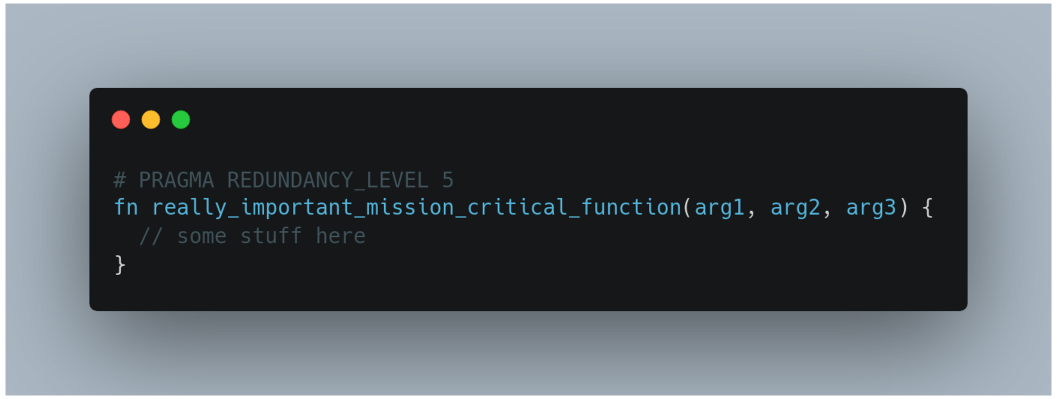
Idea: Compiling For Unreliable Hardware

- Observation: Checking results different ways tends to increase error detection and reliability.
- Idea: A sufficiently clever compiler can force the program control flow and hardware to take different paths. The chances of the same failure occurring are quite low. EQSAT!

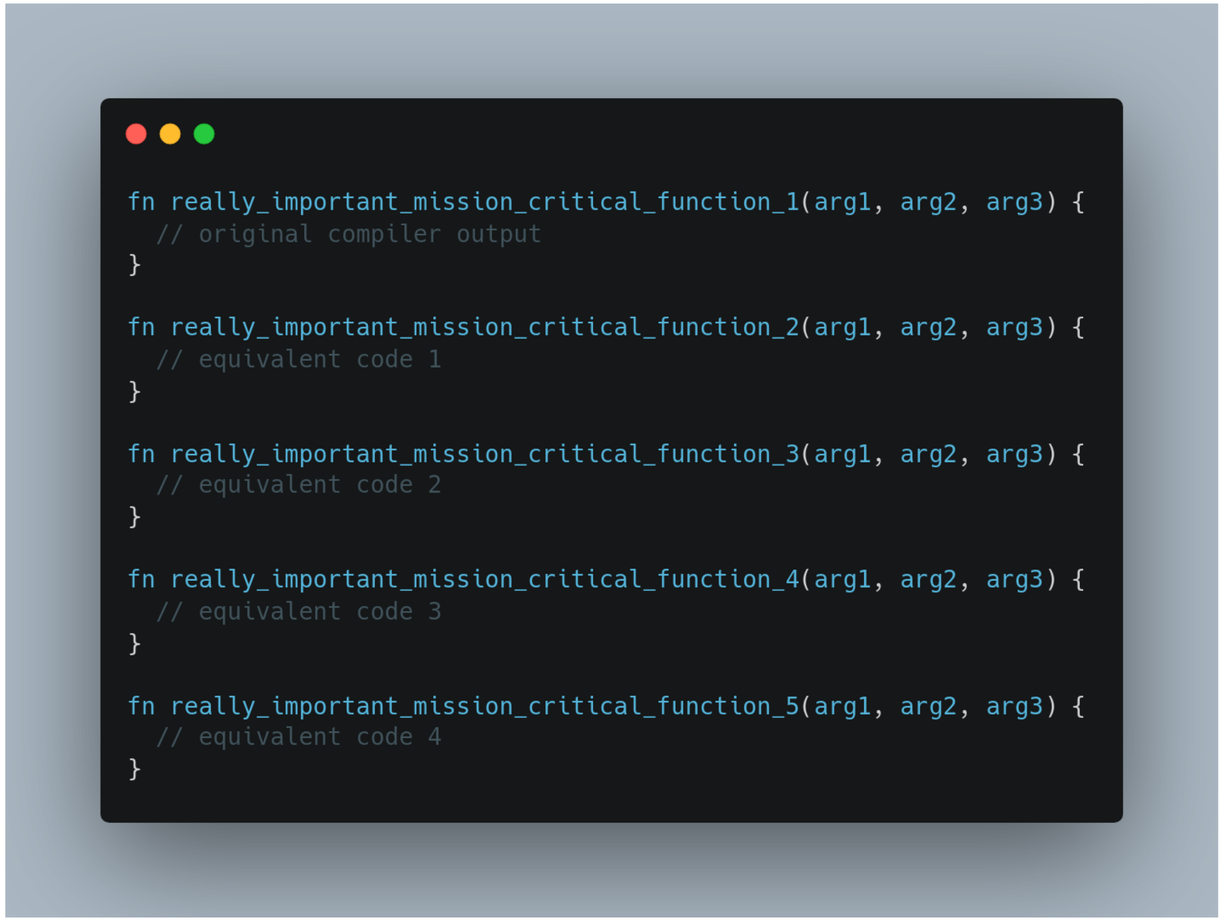
Small example

- Suppose our ALU is unreliable, but we don't know how.
- Consider:
  - let x = ((a + b) + c) + d;
  - let x = (b + c) + a + d;
  - let x = a + (b + (c + d) - 1) + 1;
- What are the chances that these all fail the same way?
- Our compiler can automatically derive these types of equivalences (and more complex ones) via EQSAT!

Example



```
# PRAGMA REDUNDANCY_LEVEL 5
fn really_important_mission_critical_function(arg1, arg2, arg3) {
  // some stuff here
}
```



```
fn really_important_mission_critical_function_1(arg1, arg2, arg3) {
  // original compiler output
}

fn really_important_mission_critical_function_2(arg1, arg2, arg3) {
  // equivalent code 1
}

fn really_important_mission_critical_function_3(arg1, arg2, arg3) {
  // equivalent code 2
}

fn really_important_mission_critical_function_4(arg1, arg2, arg3) {
  // equivalent code 3
}

fn really_important_mission_critical_function_5(arg1, arg2, arg3) {
  // equivalent code 4
}
```

Opportunities in the compiler for diversity

- Repurposing existing optimization passes:
  - Reordering of arithmetic expressions
  - Register allocation
  - LICM
  - Peephole optimization
  - Constant proprogation
  - CSE
- Other strategies:
  - EQSAT
  - Rewrite rules automatically derived from program semantics
  - Program synthesis

How do we validate?

- Simulate failures in space by running on FPGA and pinning gates high (thanks Brian!)
- Put stuff in space and collect data
- Bombard physical hardware with radiation on earth

Soundness and completeness sketch

Lecture on whiteboard:

- Introduce proof system for reasoning on traces
- Give precise notions of soundness and completeness
- Gesture at inductive proofs, maybe try to derive it live

Implementation in Copilot

Show live demo