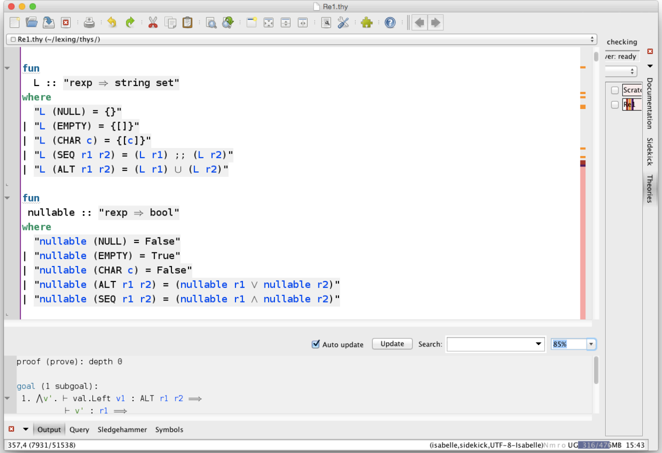
POSIX Lexing with Derivatives of Regular Expressions

Christian Urban
King's College London

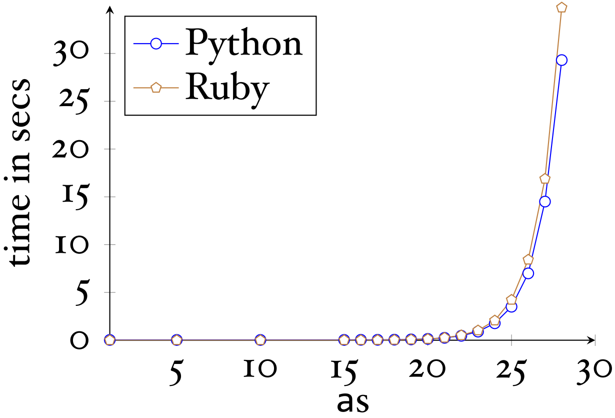
Joint work with Fahad Ausaf and Roy Dyckhoff



- Isabelle interactive theorem prover; some proofs are automatic – most however need help
- the learning curve is steep; you often have to fight the theorem prover...no different in other ITPs

Why Bother?

Surely regular expressions must have been implemented and studied to death, no?



evil regular expressions: $(a?)^n \cdot a^n$

Isabelle Theorem Prover

- started to use Isabelle after my PhD (in 2000)
- the thesis included a rather complicated "pencil-and-paper" proof for a termination argument (sort of λ-calculus)
- me, my supervisor, the examiners did not find any problems

Henk Barendregt

Andrew Pitts

- people were building their work on my result

Nominal Isabelle

- implemented a package for the Isabelle prover in order to reason conveniently about binders

$$\lambda x.\, M \qquad \forall x.\, P\, x$$

Nominal Isabelle

- implemented a package for the Isabelle prover in order to reason conveniently about binders

$$\lambda x.\, M \qquad \forall x.\, P\, x$$

Nominal Isabelle

- implemented a package for the Isabelle prover in order to reason conveniently about binders

$$\lambda x.\, M \qquad \forall x.\, P\, x$$

- when finally being able to formalise the proof from my PhD, I found that the main result (termination) is correct, but a central lemma needed to be generalised

Variable Convention

Variable Convention:

If $M_1, \ldots, M_n$ occur in a certain mathematical context (e.g. definition, proof), then in these terms all bound variables are chosen to be different from the free variables.

Barendregt in "The Lambda-Calculus: Its Syntax and Semantics"

- instead of proving a property for **all** bound variables, you prove it only for **some**...?
- feels like it is used in 90% of papers in PT and FP (9.9% use de-Bruijn indices)
- this is mostly OK, but in some corner-cases you can use it to prove **false**...we fixed this!

Bob Harper

Frank Pfenning

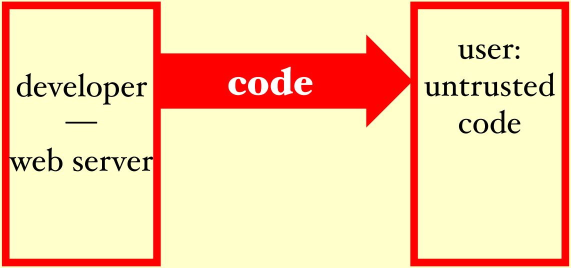
published a proof in **ACM Transactions on Computational Logic**, 2005, ∼31pp

Andrew Appel

relied on their proof in a **security** critical application

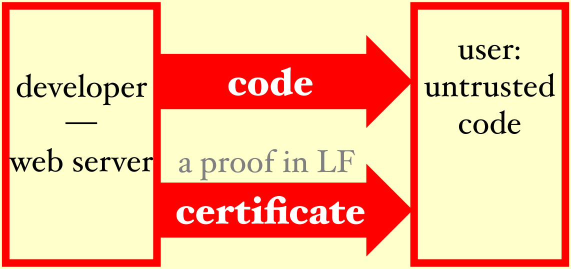
Proof-Carrying Code

Idea:



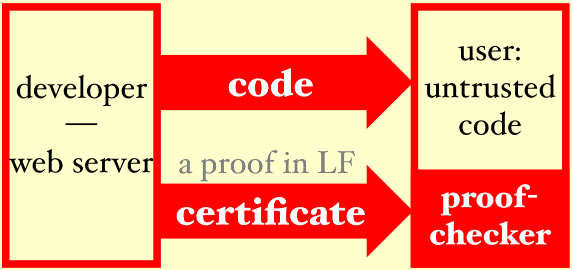
Proof-Carrying Code

Idea:



Proof-Carrying Code

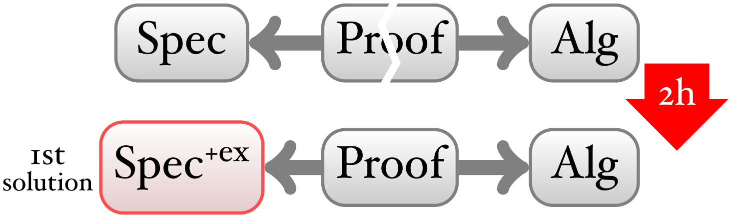
Idea:

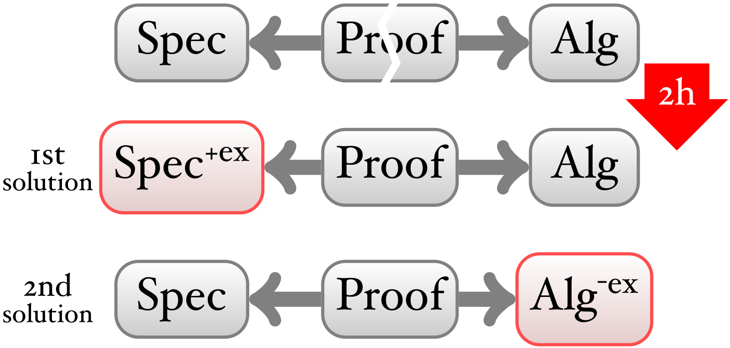


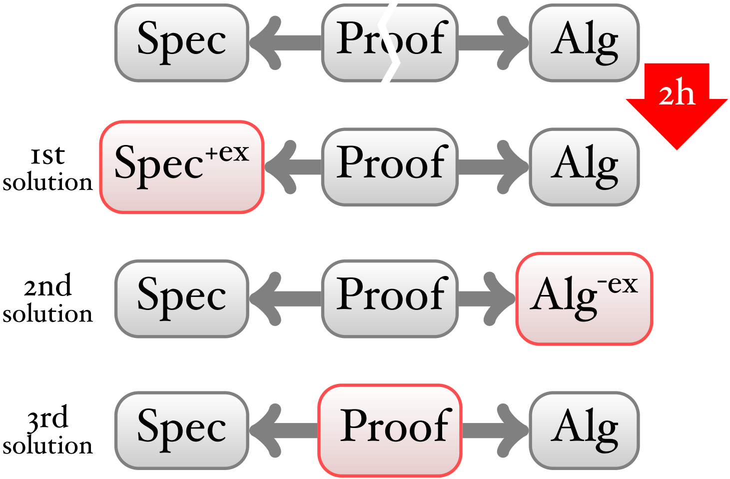
- Appel's checker is ∼2700 lines of code (1865 loc of LF definitions; 803 loc in C including 2 library functions)
- 167 loc in C implement a type-checker


Spec
Proof
Alg


Spec
Proof
Alg

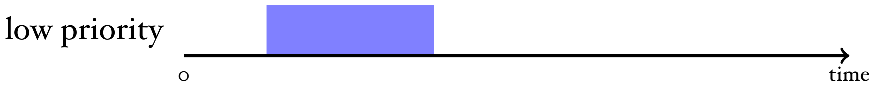

Spec
Proof
Alg
2h
1st solution
Spec+ex
Proof
Alg


Spec
Proof
Alg
2h
1st solution
Spec+ex
Proof
Alg
2nd solution
Spec
Proof
Alg-ex

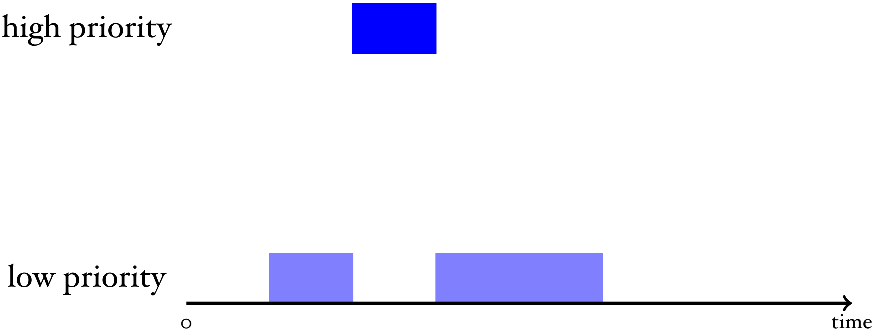


Each time one needs to check ∼31pp of informal paper proofs. You have to be able to keep definitions and proofs consistent.

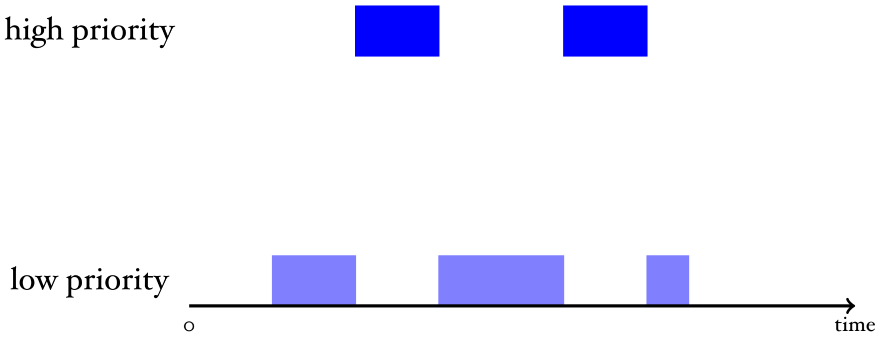
Real-Time Scheduling



Real-Time Scheduling

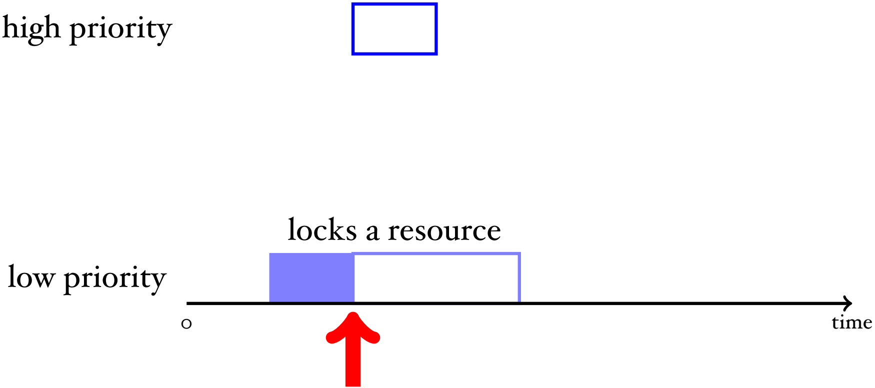


Real-Time Scheduling



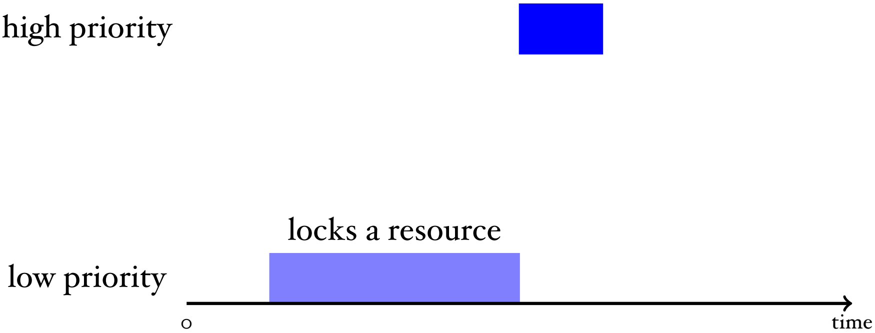
RT-Scheduling: You want to avoid that a high-priority process is starved indefinitely.

Real-Time Scheduling



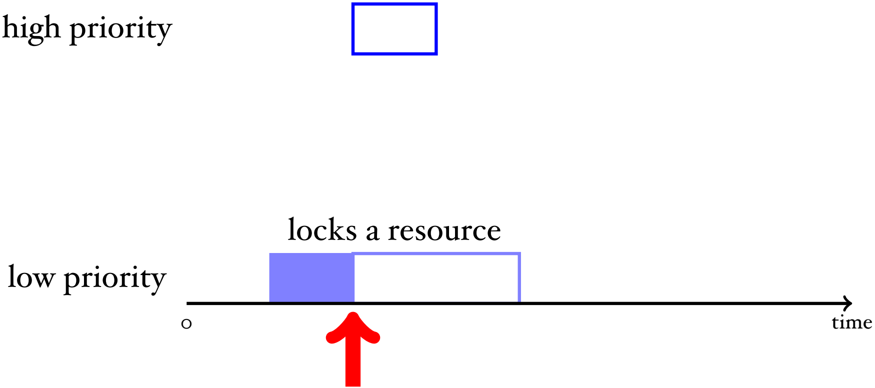
RT-Scheduling: You want to avoid that a high-priority process is starved indefinitely.

Real-Time Scheduling



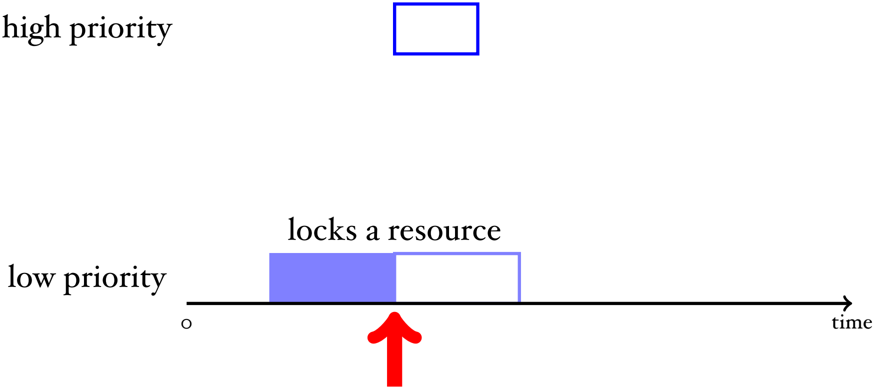
RT-Scheduling: You want to avoid that a high-priority process is starved indefinitely.

Real-Time Scheduling



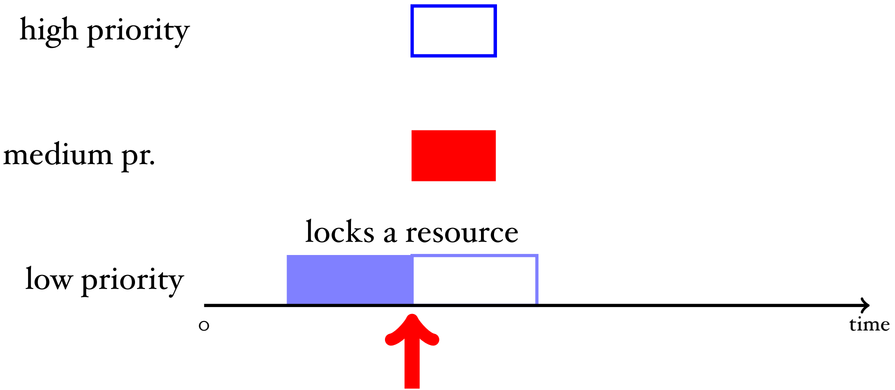
RT-Scheduling: You want to avoid that a high-priority process is starved indefinitely.

Real-Time Scheduling



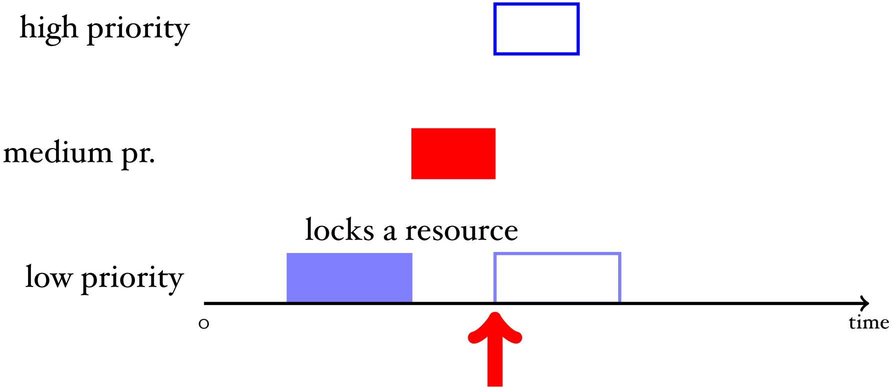
RT-Scheduling: You want to avoid that a high-priority process is starved indefinitely.

Real-Time Scheduling



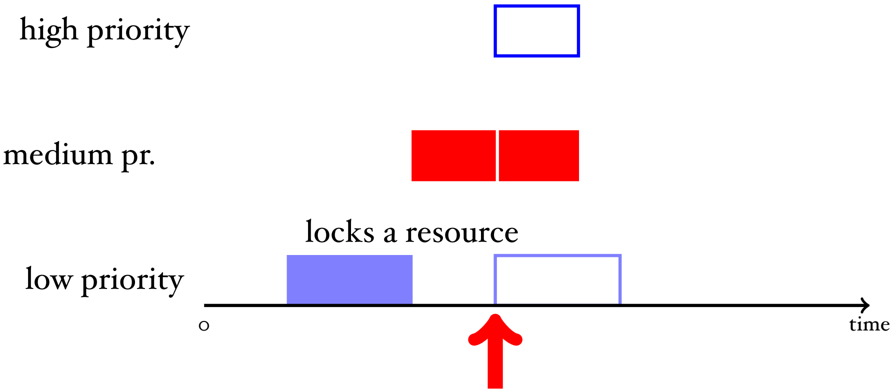
RT-Scheduling: You want to avoid that a high-priority process is starved indefinitely.

Real-Time Scheduling



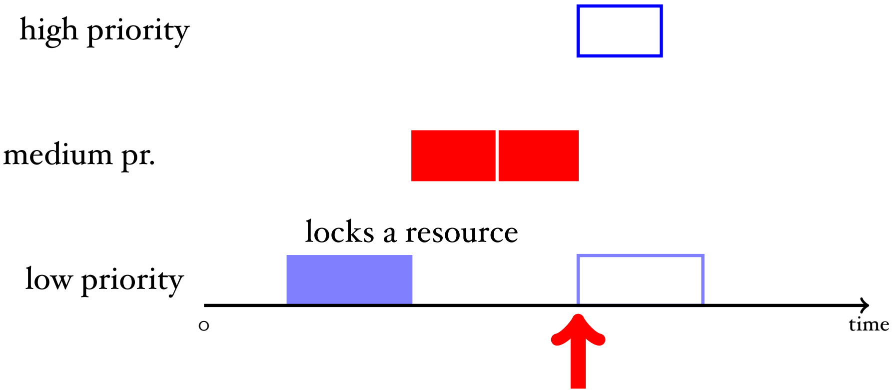
RT-Scheduling: You want to avoid that a high-priority process is starved indefinitely.

Real-Time Scheduling



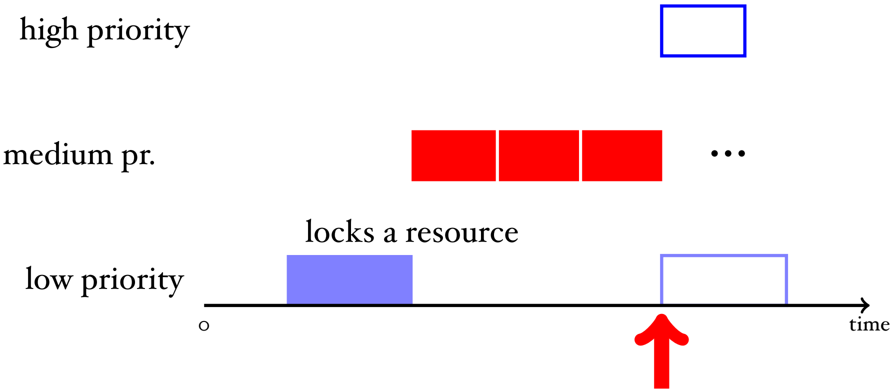
RT-Scheduling: You want to avoid that a high-priority process is starved indefinitely.

Real-Time Scheduling



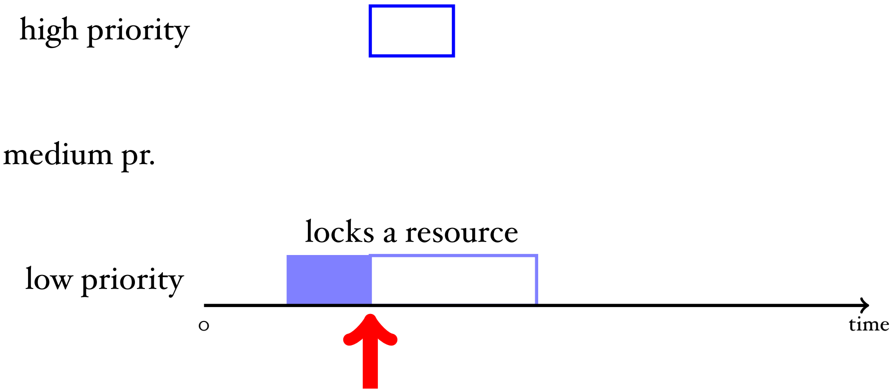
RT-Scheduling: You want to avoid that a high-priority process is starved indefinitely.

Real-Time Scheduling



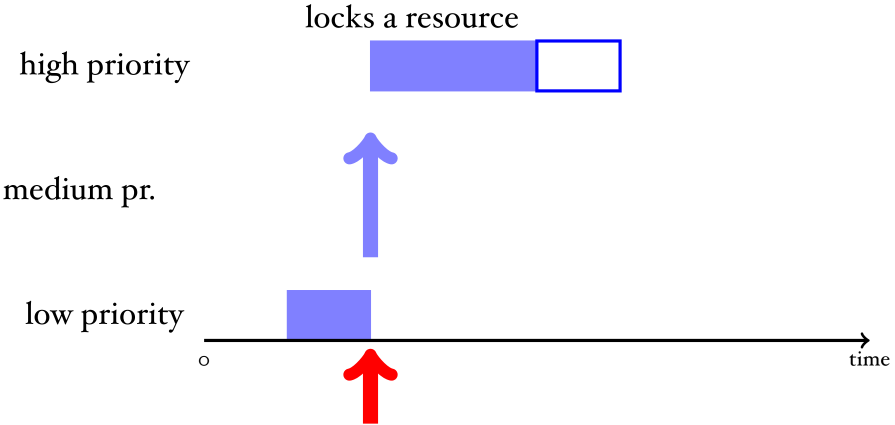
RT-Scheduling: You want to avoid that a high-priority process is starved indefinitely.

Real-Time Scheduling



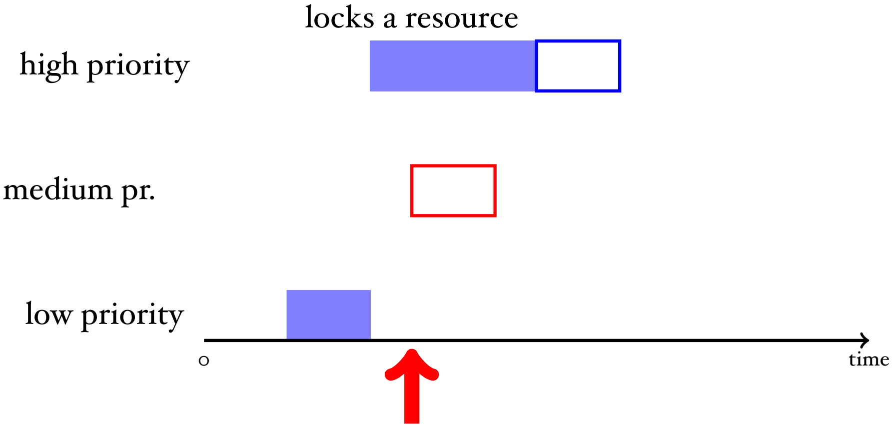
RT-Scheduling: You want to avoid that a high-priority process is starved indefinitely.

Real-Time Scheduling



RT-Scheduling: You want to avoid that a high-priority process is starved indefinitely.

Real-Time Scheduling



RT-Scheduling: You want to avoid that a high-priority process is starved indefinitely.

Priority Inheritance Scheduling

- Let a low priority process L temporarily inherit the high priority of H until L leaves the critical section unlocking the resource.

- Once the resource is unlocked, L "returns to its original priority level."

L. Sha, R. Rajkumar, and J. P. Lehoczky. *Priority Inheritance Protocols: An Approach to Real-Time Synchronization*. IEEE Transactions on Computers, 39(9):1175–1185, 1990

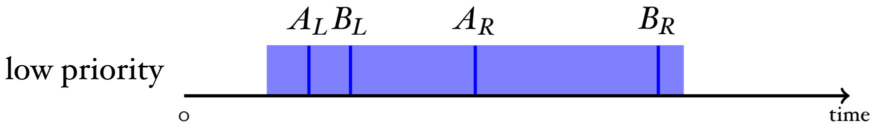
Priority Inheritance Scheduling

- Let a low priority process L temporarily inherit the high priority of H until L leaves the critical section unlocking the resource.

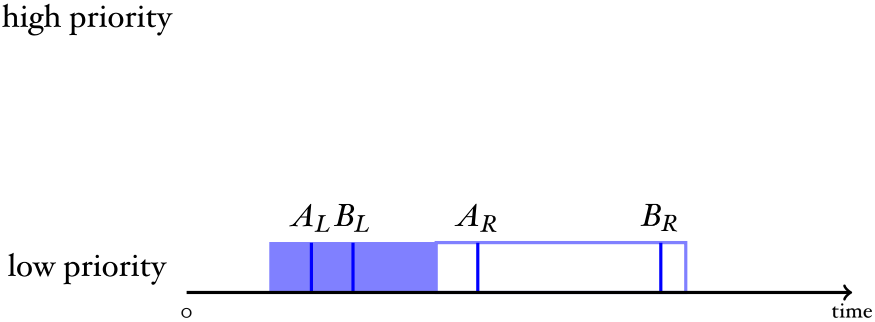
- Once the resource is unlocked, L “returns to its original priority level.”

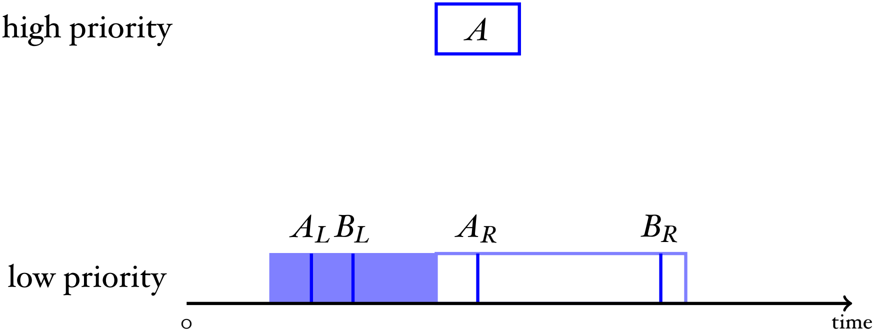
  L. Sha, R. Rajkumar, and J. P. Lehoczky. *Priority Inheritance Protocols: An Approach to Real-Time Synchronization*. IEEE Transactions on Computers, 39(9):1175–1185, 1990

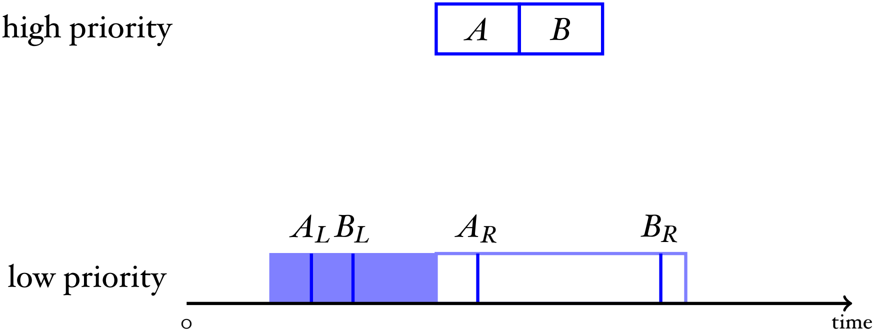
- Proved correct, reviewed in a respectable journal....what could possibly be wrong?

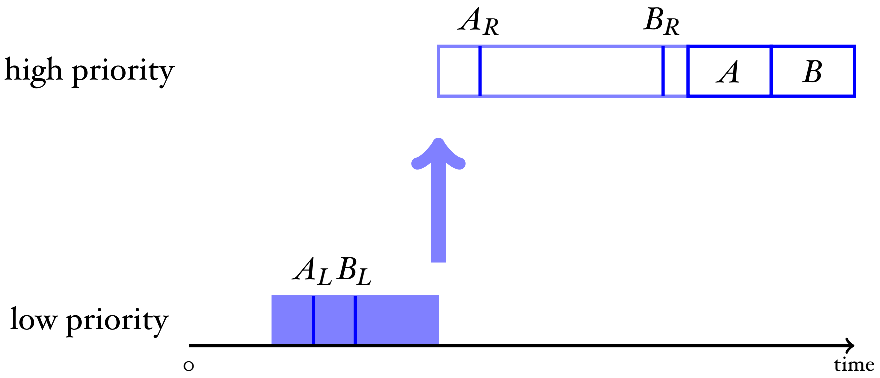


A_L B_L
A_R
B_R
low priority
0
time

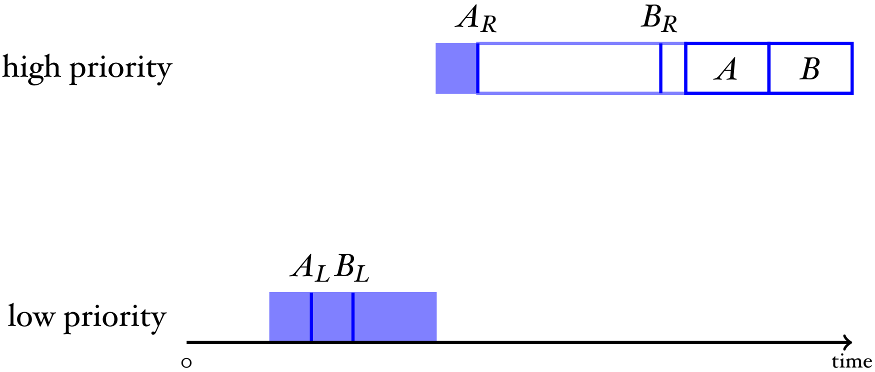

high priority
low priority
A_L B_L
A_R
B_R
0
time


high priority
A
low priority
A_L B_L
A_R
B_R
0
time

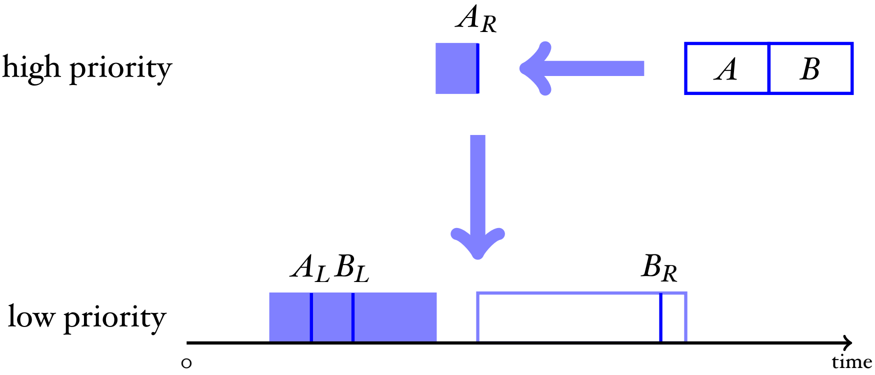

high priority
A
B
low priority
A_L B_L
A_R
B_R
0
time



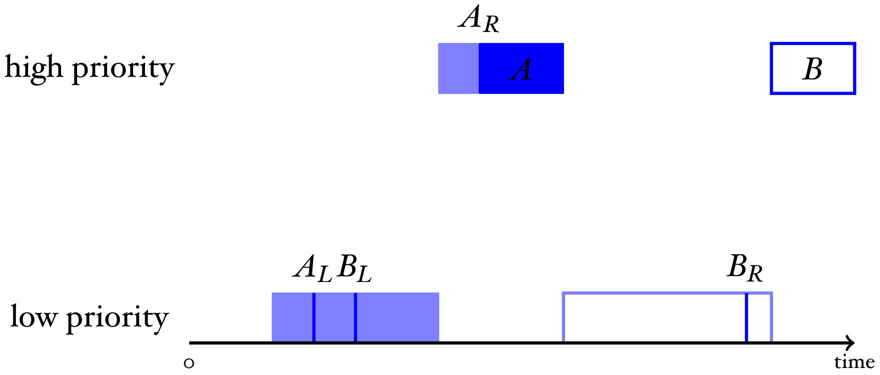
high priority
A_R
B_R
A
B
A_L B_L
low priority
0
time

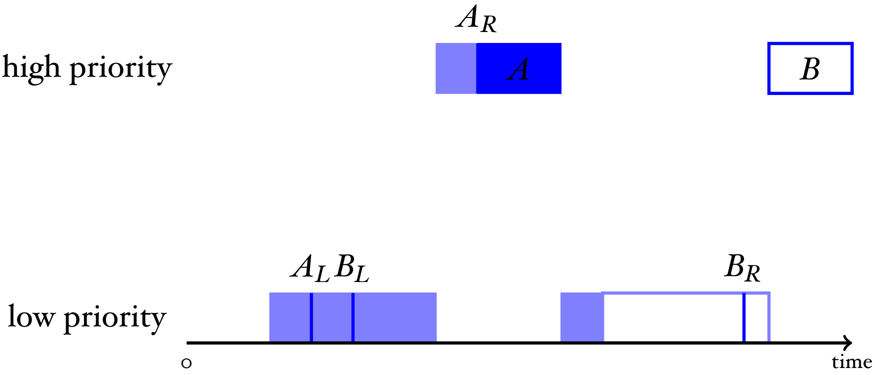


high priority
low priority
A_R
B_R
A
B
A_L B_L
0
time

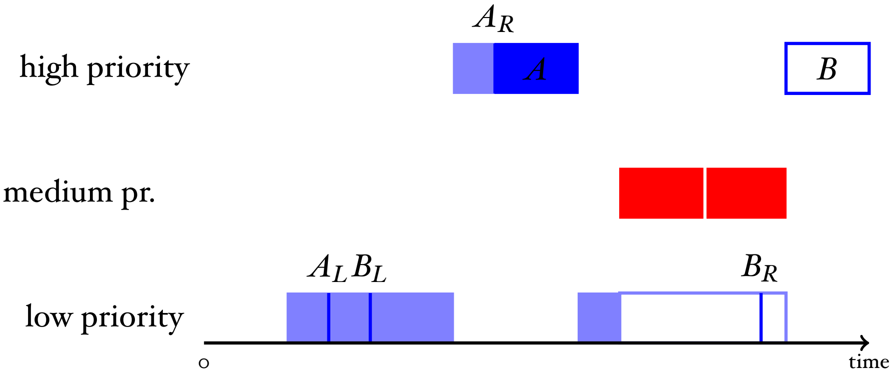


high priority
A_R
A
B
low priority
A_L B_L
B_R
0
time

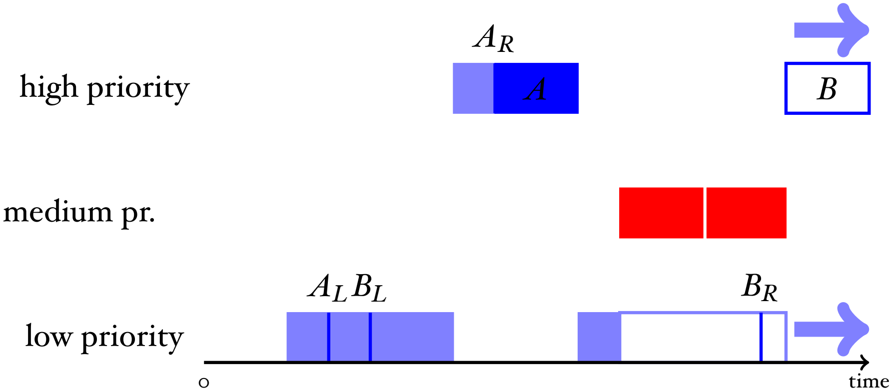

A_R
high priority
A
B
A_L B_L
B_R
low priority
0
time



A_R
high priority
A
B
A_L B_L
B_R
low priority
0
time



A_R
high priority
A
B
medium pr.
A_L B_L
B_R
low priority
0
time



Scheduling: You want to avoid that a high priority process is starved indefinitely.

Priority Inheritance Scheduling

- Let a low priority process L temporarily inherit the high priority of H until L leaves the critical section unlocking the resource.

- Once the resource is unlocked, L returns to its original priority level. **BOGUS**

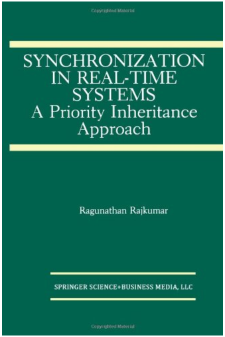
Priority Inheritance Scheduling

- Let a low priority process *L* temporarily inherit the high priority of *H* until *L* leaves the critical section unlocking the resource.

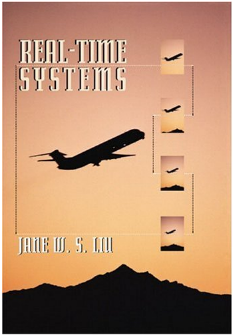
- Once the resource is unlocked, *L* returns to its original priority level. **BOGUS**

- ...*L* needs to switch to the highest **remaining** priority of the threads that it blocks.

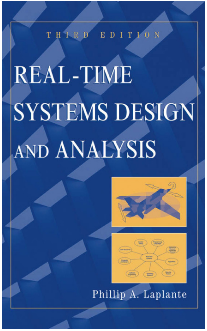
this error is already known since around 1999



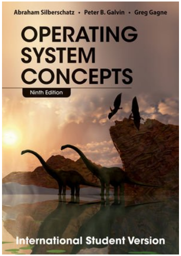
- by Rajkumar, 1991
- *"it resumes the priority it had at the point of entry into the critical section"*



- by Jane Liu, 2000
- *"The job J_l executes at its inherited priority until it releases R; at that time, the priority of J_l returns to its priority at the time when it acquires the resource R."*
- gives pseudo code and totally bogus data structures
- interesting part is "*left as an exercise*"



- by Laplante and Ovaska, 2011 ($113.76)
- *"when [the task] exits the critical section that caused the block, it reverts to the priority it had when it entered that section"*



- by Silberschatz, Galvin and Gagne (9th edition, 2013)
- *"Upon releasing the lock, the [low-priority] thread will revert to its original priority."*

Priority Scheduling

- a scheduling algorithm that is widely used in real-time operating systems
- has been “proved” correct by hand in a paper in 1990
- but this algorithm turned out to be incorrect, despite its “proof”

Priority Scheduling

- a scheduling algorithm that is widely used in real-time operating systems
- has been "proved" correct by hand in a paper in 1990
- but this algorithm turned out to be incorrect, despite its "proof"

- we (generalised) the algorithm and then **really** proved that it is correct
- we implemented this algorithm in a small OS called PINTOS (used for teaching at Stanford)
- our implementation was faster than their reference implementation

Lessons Learned

- our proof-technique is adapted from security protocols
- do not venture outside your field of expertise
- we solved the single-processor case; the multi-processor case: no idea!

Regular Expressions

r	$::=$	$\varnothing$	null
	$\mid$	ϵ	empty string
	$\mid$	c	character
	$\mid$	$r_1 \cdot r_2$	sequence
	$\mid$	$r_1 + r_2$	alternative / choice
	$\mid$	r^*	star (zero or more)

The Derivative of a Rexp

If r matches the string $c :: s$, what is a regular expression that matches just s?

$der\ c\ r$ gives the answer, Brzozowski (1964), Owens (2005) "...have been lost in the sands of time..."

...whether a regular expression can match the empty string:

$$
\begin{array}{lcl}
\mathit{nullable}(\varnothing) & \stackrel{\text{def}}{=} & \mathit{false} \\
\mathit{nullable}(\epsilon) & \stackrel{\text{def}}{=} & \mathit{true} \\
\mathit{nullable}(c) & \stackrel{\text{def}}{=} & \mathit{false} \\
\mathit{nullable}(r_1 + r_2) & \stackrel{\text{def}}{=} & \mathit{nullable}(r_1) \vee \mathit{nullable}(r_2) \\
\mathit{nullable}(r_1 \cdot r_2) & \stackrel{\text{def}}{=} & \mathit{nullable}(r_1) \wedge \mathit{nullable}(r_2) \\
\mathit{nullable}(r^*) & \stackrel{\text{def}}{=} & \mathit{true}
\end{array}
$$

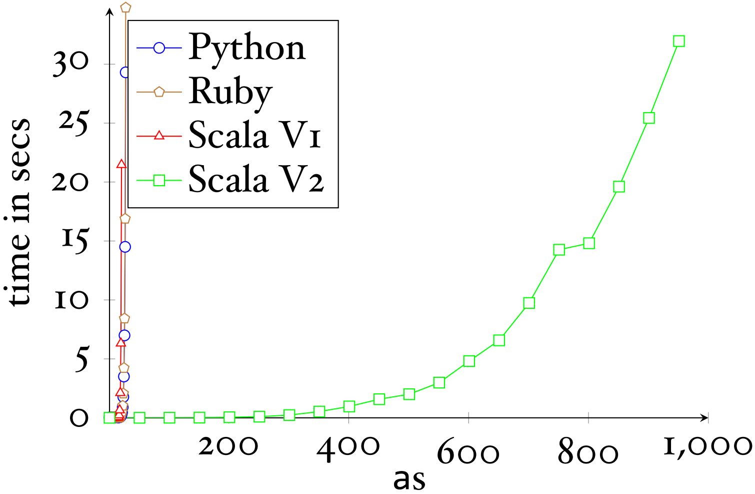
The Derivative of a Rexp

$$
\begin{array}{lcl}
der\,c\,(\varnothing) & \stackrel{\text{def}}{=} & \varnothing \\
der\,c\,(\epsilon) & \stackrel{\text{def}}{=} & \varnothing \\
der\,c\,(d) & \stackrel{\text{def}}{=} & \text{if } c = d \text{ then } \epsilon \text{ else } \varnothing \\
der\,c\,(r_1 + r_2) & \stackrel{\text{def}}{=} & der\,c\,r_1 + der\,c\,r_2 \\
der\,c\,(r_1 \cdot r_2) & \stackrel{\text{def}}{=} & \text{if } nullable(r_1) \\
 & & \text{then } (der\,c\,r_1) \cdot r_2 + der\,c\,r_2 \\
 & & \text{else } (der\,c\,r_1) \cdot r_2 \\
der\,c\,(r^*) & \stackrel{\text{def}}{=} & (der\,c\,r) \cdot (r^*)
\end{array}
$$

The Derivative of a Rexp

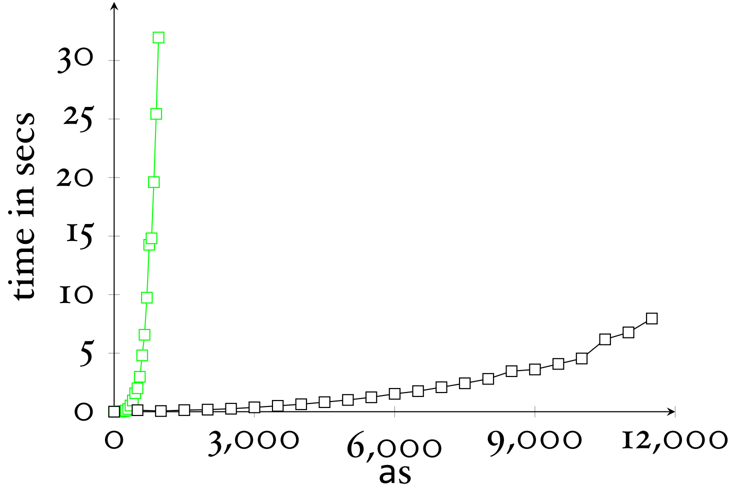
$$
\begin{array}{lcl}
der\,c\,(\varnothing) & \stackrel{\text{def}}{=} & \varnothing \\
der\,c\,(\epsilon) & \stackrel{\text{def}}{=} & \varnothing \\
der\,c\,(d) & \stackrel{\text{def}}{=} & \text{if } c = d \text{ then } \epsilon \text{ else } \varnothing \\
der\,c\,(r_1 + r_2) & \stackrel{\text{def}}{=} & der\,c\,r_1 + der\,c\,r_2 \\
der\,c\,(r_1 \cdot r_2) & \stackrel{\text{def}}{=} & \text{if } nullable(r_1) \\
 & & \text{then } (der\,c\,r_1) \cdot r_2 + der\,c\,r_2 \\
 & & \text{else } (der\,c\,r_1) \cdot r_2 \\
der\,c\,(r^*) & \stackrel{\text{def}}{=} & (der\,c\,r) \cdot (r^*) \\
 & & \\
ders\,[]\,r & \stackrel{\text{def}}{=} & r \\
ders\,(c::s)\,r & \stackrel{\text{def}}{=} & ders\,s\,(der\,c\,r)
\end{array}
$$

$(a?)^n \cdot a^n$



Python
Ruby
Scala V1
Scala V2
time in secs
0
5
10
15
20
25
30
200
400
600
800
1,000
as

$(a?)^n \cdot a^n$



Correctness

It is a relative easy exercise in a theorem prover:

$$matches(r, s) \text{ if and only if } s \in L(r)$$

$$matches(r, s) \stackrel{\text{def}}{=} nullable(ders(r, s))$$

POSIX Regex Matching

Two rules:

- Longest match rule ("maximal munch rule"): The longest initial substring matched by any regular expression is taken as the next token.

i	f	f	o	o	␣	b	l	a

- Rule priority: For a particular longest initial substring, the first regular expression that can match determines the token.

i	f	␣	b	l	a

POSIX Regex Matching

Two rules:

- Longest match rule ("maximal munch rule"): The longest initial substring matched by any regular expression is taken as the next token.

i	f	f	o	o	␣	b	l	a

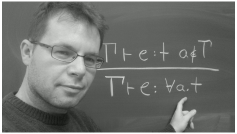
- Rule priority: For a particular longest initial substring, the first regular expression that can match determines the token.

i	f	␣	b	l	a

Kuklewicz: most POSIX matchers are buggy
http://www.haskell.org/haskellwiki/Regex_Posix

POSIX Regex Matching

- Sulzmann & Lu came up with a beautiful idea for how to extend the simple regular expression matcher to POSIX matching/lexing (FLOPS 2014)



Martin Sulzmann

- the idea: define an inverse operation to the derivatives

Regexes and Values

Regular expressions and their corresponding values:

$$
\begin{array}{rcl}
r & ::= & \varnothing \\
  & \mid & \epsilon \\
  & \mid & c \\
  & \mid & r_1 \cdot r_2 \\
  & \mid & r_1 + r_2 \\
  & & \\
  & \mid & r^*
\end{array}
\qquad
\begin{array}{rcl}
v & ::= & \\
  & & \mathit{Empty} \\
  & \mid & \mathit{Char}(c) \\
  & \mid & \mathit{Seq}(v_1, v_2) \\
  & \mid & \mathit{Left}(v) \\
  & \mid & \mathit{Right}(v) \\
  & \mid & [] \\
  & \mid & [v_1, \ldots\ v_n]
\end{array}
$$

Regexes and Values

Regular expressions and their corresponding values:

$$
\begin{array}{rcl}
r & ::= & \varnothing \\
  & | & \epsilon \\
  & | & c \\
  & | & r_1 \cdot r_2 \\
  & | & r_1 + r_2 \\
  & & \\
  & | & r^*
\end{array}
\qquad
\begin{array}{rcl}
v & ::= & \\
  & & Empty \\
  & | & Char(c) \\
  & | & Seq(v_1, v_2) \\
  & | & Left(v) \\
  & | & Right(v) \\
  & | & [] \\
  & | & [v_1, \ldots v_n]
\end{array}
$$

There is also a notion of a string behind a value: $|v|$

Sulzmann & Lu Matcher

We want to match the string *abc* using r_1:

$$r_1 \xrightarrow{der\ a} r_2$$

Sulzmann & Lu Matcher

We want to match the string *abc* using r_1:

$$r_1 \xrightarrow{der\,a} r_2 \xrightarrow{der\,b} r_3$$

Sulzmann & Lu Matcher

We want to match the string abc using r_1:

$$r_1 \xrightarrow{der\,a} r_2 \xrightarrow{der\,b} r_3 \xrightarrow{der\,c} r_4$$

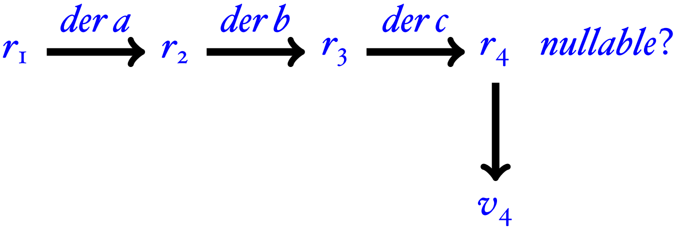
Sulzmann & Lu Matcher

We want to match the string *abc* using r_1:

$$r_1 \xrightarrow{der\,a} r_2 \xrightarrow{der\,b} r_3 \xrightarrow{der\,c} r_4 \quad nullable?$$

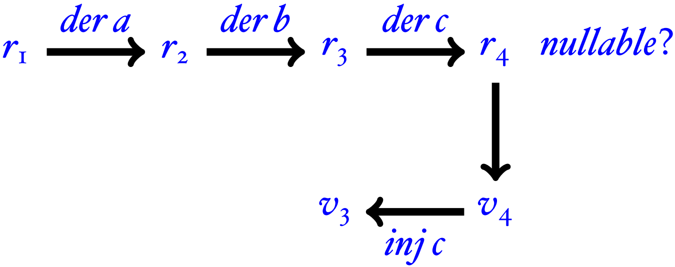
Sulzmann & Lu Matcher

We want to match the string *abc* using r_1:



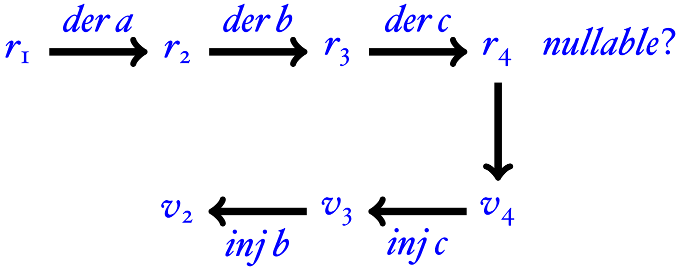
Sulzmann & Lu Matcher

We want to match the string *abc* using r_1:



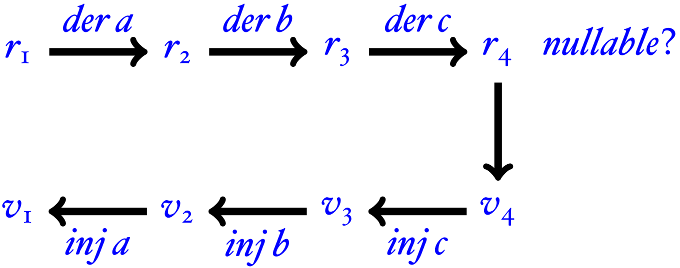
Sulzmann & Lu Matcher

We want to match the string *abc* using r_1:



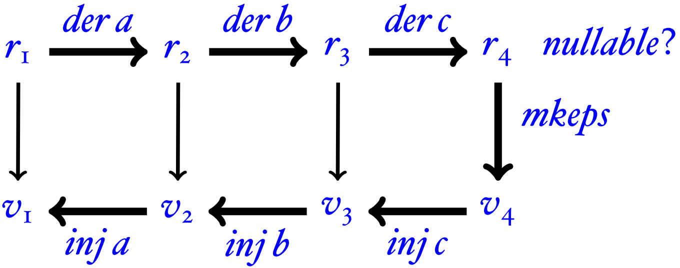
Sulzmann & Lu Matcher

We want to match the string *abc* using r_1:



Sulzmann & Lu Matcher

We want to match the string *abc* using r_1:



Sulzmann & Lu Paper

- I have no doubt the algorithm is correct — the problem, I do not believe their proof.

> "How could I miss this? Well, I was rather careless when stating this Lemma :)
>
> Great example how formal machine checked proofs (and proof assistants) can help to spot flawed reasoning steps."

Sulzmann & Lu Paper

- I have no doubt the algorithm is correct — the problem, I do not believe their proof.

> "How could I miss this? Well, I was rather careless when stating this Lemma :)
>
> Great example how formal machine checked proofs (and proof assistants) can help to spot flawed reasoning steps."

> "Well, I don't think there's any flaw. The issue is how to come up with a mechanical proof. In my world mathematical proof = mechanical proof doesn't necessarily hold."

Sulzmann & Lu Paper

- I have no doubt the algorithm is correct — the problem, I do not believe their proof.

"How could I miss this? Well, I was rather careless

Lemma 3 (Projection and Injection). *Let r be a regular expression, l a letter and v a parse tree.*

1. *If $\vdash v : r$ and $|v| = lw$ for some word w, then $\vdash proj_{(r,l)}\ v : r\backslash l$.*
2. *If $\vdash v : r\backslash l$ then $(proj_{(r,l)} \circ inj_{r\backslash l})\ v = v$.*
3. *If $\vdash v : r$ and $|v| = lw$ for some word w, then $(inj_{r\backslash l} \circ proj_{(r,l)})\ v = v$.*

MS:BUG[Come accross this issue when going back to our constructive reg-ex work] Consider $\vdash [Right\ (), Left\ a] : (a + \epsilon)^*$. However, $proj_{((a+\epsilon)^*,a)}\ [Right\ (), Left\ a]$ fails! The point is that *proj* only works correctly if applied on POSIX parse trees.

MS:Possible fixes We only ever apply *proj* on Posix parse trees.

For convenience, we write "$\vdash v : r$ is POSIX" where we mean that $\vdash v : r$ holds and v is the POSIX parse tree of r for word $|v|$.

Lemma 2 follows from the following statement.

necessarily hold."

The Proof Idea by Sulzmann & Lu

- introduce an inductively defined ordering relation $v \succ_r v'$ which captures the idea of POSIX matching
- the algorithm returns the maximum of all possible values that are possible for a regular expression.

The Proof Idea by Sulzmann & Lu

- introduce an inductively defined ordering relation $v \succ_r v'$ which captures the idea of POSIX matching
- the algorithm returns the maximum of all possible values that are possible for a regular expression.

- the idea is from a paper by Cardelli & Frisch about greedy matching (greedy = preferring instant gratification to delayed repletion):
- e.g. given $(a + (b + ab))^*$ and string ab

greedy:	$[Left(a), Right(Left(b)]$
POSIX:	$[Right(Right(a, b)))]$

$$\frac{}{\vdash Empty : \epsilon} \qquad \frac{}{\vdash Char(c) : c}$$

$$\frac{\vdash v_1 : r_1 \quad \vdash v_2 : r_2}{\vdash Seq(v_1, v_2) : r_1 \cdot r_2}$$

$$\frac{\vdash v : r_1}{\vdash Left(v) : r_1 + r_2} \qquad \frac{\vdash v : r_2}{\vdash Right(v) : r_1 + r_2}$$

$$\frac{}{\vdash [] : r^*} \qquad \frac{\vdash v_1 : r \quad \ldots \quad \vdash v_n : r}{\vdash [v_1, \ldots, v_n] : r^*}$$

$$POSIX(v,r) \stackrel{\text{def}}{=} \vdash v : r \wedge (\forall v'.\ \vdash v' : r \wedge |v'| = |v| \Rightarrow v \succ_r v')$$

$$\frac{v_1 = v'_1 \quad v_2 \succ_{r_2} v'_2}{Seq(v_1, v_2) \succ_{r_1 \cdot r_2} Seq(v'_1, v'_2)} \qquad \frac{v_1 \neq v'_1 \quad v_1 \succ_{r_1} v'_1}{Seq(v_1, v_2) \succ_{r_1 \cdot r_2} Seq(v'_1, v'_2)}$$

$$\frac{v \succ_{r_1} v'}{Left(v) \succ_{r_1 + r_2} Left(v')} \qquad \frac{v \succ_{r_2} v'}{Right(v) \succ_{r_1 + r_2} Right(v')}$$

$$\frac{length|v| \geq length|v'|}{Left(v) \succ_{r_1 + r_2} Right(v')} \qquad \frac{length|v| > length|v'|}{Right(v) \succ_{r_1 + r_2} Left(v')}$$

$$\ldots$$

Problems

- Sulzmann: ...Let's assume v is not a *POSIX* value, then there must be another one ...contradiction.

Problems

- Sulzmann: ...Let's assume v is not a *POSIX* value, then there must be another one ...contradiction.
- Exists?

$$L(r) \neq \varnothing \;\Rightarrow\; POSIX(v, r)$$

Problems

- Sulzmann: ...Let's assume v is not a *POSIX* value, then there must be another one ...contradiction.

- Exists?

$$L(r) \neq \varnothing \;\Rightarrow\; POSIX(v, r)$$

- in the sequence case, the induction hypotheses require $|v_1| = |v'_1|$ and $|v_2| = |v'_2|$, but you only know

$$|v_1| \mathbin{@} |v_2| = |v'_1| \mathbin{@} |v'_2|$$

Problems

- Sulzmann: ...Let's assume v is not a *POSIX* value, then there must be another one ...contradiction.

- Exists?

$$L(r) \neq \varnothing \;\Rightarrow\; POSIX(v, r)$$

- in the sequence case, the induction hypotheses require $|v_1| = |v'_1|$ and $|v_2| = |v'_2|$, but you only know

$$|v_1| \mathbin{@} |v_2| = |v'_1| \mathbin{@} |v'_2|$$

- although one begins with the assumption that the two values have the same flattening, this cannot be maintained as one descends into the induction (alternative, sequence)

Our Solution

- direct definition of what a POSIX value is, using $s \in r \to v$:

$$\frac{}{[] \in \epsilon \to \mathit{Empty}} \qquad \frac{}{c \in c \to \mathit{Char}(c)}$$

$$\frac{s \in r_1 \to v}{s \in r_1 + r_2 \to \mathit{Left}(v)} \qquad \frac{s \in r_2 \to v \quad s \notin L(r_1)}{s \in r_1 + r_2 \to \mathit{Right}(v)}$$

$$\frac{\begin{array}{l} s_1 \in r_1 \to v_1 \\ s_2 \in r_2 \to v_2 \\ \neg(\exists s_3\, s_4.\, s_3 \neq [] \wedge s_3 @ s_4 = s_2 \wedge s_1 @ s_3 \in L(r_1) \wedge s_4 \in L(r_2)) \end{array}}{s_1 @ s_2 \in r_1 \cdot r_2 \to \mathit{Seq}(v_1, v_2)}$$

...

Pencil-and-Paper Proofs in CS are normally incorrect

- case in point, in one of Roy's proofs he made the incorrect inference

  if $\forall s.\ |v_2| \notin L(der\,c\,r_1) \cdot s$ then $\forall s.\ c\,|v_2| \notin L(r_1) \cdot s$

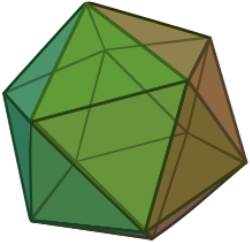
  while

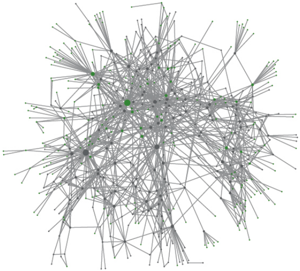
  if $\forall s.\ |v_2| \in L(der\,c\,r_1) \cdot s$ then $\forall s.\ c\,|v_2| \in L(r_1) \cdot s$

  is correct

Proofs in Math vs. in CS

My theory on why CS-proofs are often buggy





Math:

in math, "objects" can be "looked" at from all "angles";

non-trivial proofs, but it seems difficult to make mistakes

Code in CS:

the call-graph of the seL4 microkernel OS;

easy to make mistakes

Conclusion

- we strengthened the POSIX definition of Sulzmann & Lu in order to get the inductions through, his proof contained small gaps but had also fundamental flaws

- its a nice exercise for theorem proving
- some optimisations need to be aplied to the algorithm in order to become fast enough
- can be used for lexing, small little functional program

Thank you very much again for the invitation!

Questions?