

# POSIX Lexing with Derivatives of Regular Expressions

Christian Urban  
King's College London

Joint work with Fahad Ausaf and Roy Dyckhoff

```
Re1.thy
Re1.thy (~/.lexing/thys/)
checking
ver: ready
Documentation
Sledgekick
Theories

fun
  L :: "rexp => string set"
where
  "L (NULL) = {}"
| "L (EMPTY) = {[]}"
| "L (CHAR c) = {[c]}"
| "L (SEQ r1 r2) = (L r1) ;; (L r2)"
| "L (ALT r1 r2) = (L r1) U (L r2)"

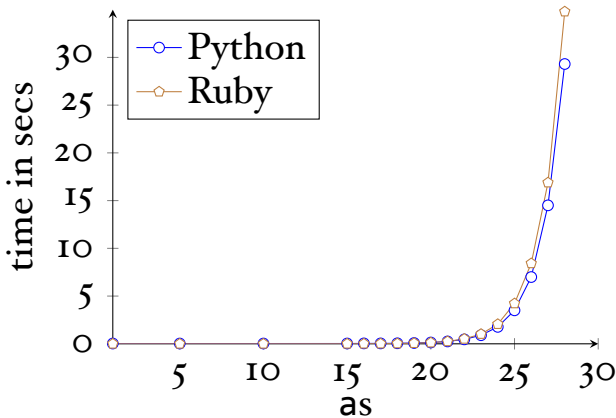
fun
  nullable :: "rexp => bool"
where
  "nullable (NULL) = False"
| "nullable (EMPTY) = True"
| "nullable (CHAR c) = False"
| "nullable (ALT r1 r2) = (nullable r1 ∨ nullable r2)"
| "nullable (SEQ r1 r2) = (nullable r1 ∧ nullable r2)"

proof (prove): depth 0
goal (1 subgoal):
1.  $\forall v'. \vdash \text{val.Left } v1 : \text{ALT } r1 \ r2 \implies$ 
    $\vdash v' : r1 \implies$ 
Output Query Sledgehammer Symbols
357.4 (7931/51538) @isabelle,sidekick.UTF-8-Isabelle)N/m / UC 15:43
```

- 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  $\lambda$ -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

$\lambda x. M$



$\forall x. P x$



# Nominal Isabelle

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

$\lambda x. M$



$\forall x. P x$



# Nominal Isabelle

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

$\lambda x. M$

$\forall x. Px$



- 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  $M_1, \dots, M_n$  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-Brujin 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

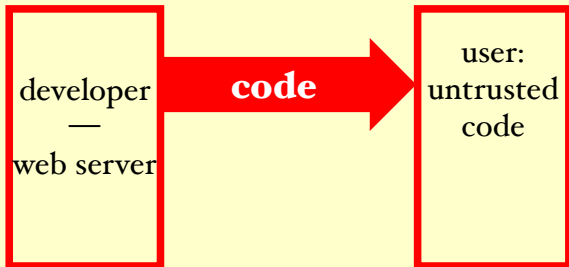


Andrew Appel

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

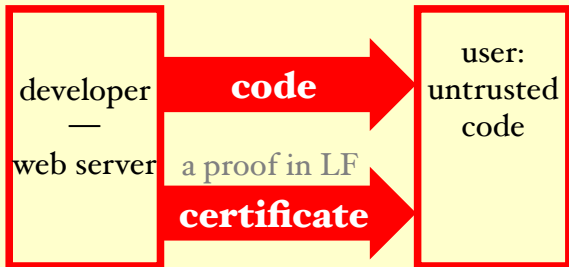
# Proof-Carrying Code

Idea:



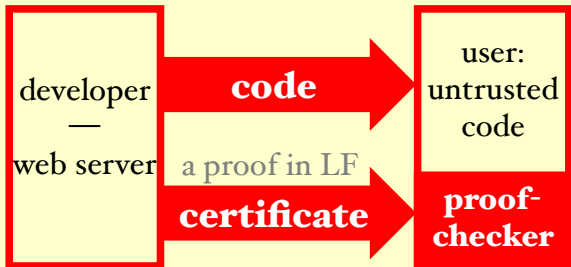
# Proof-Carrying Code

Idea:



# Proof-Carrying Code

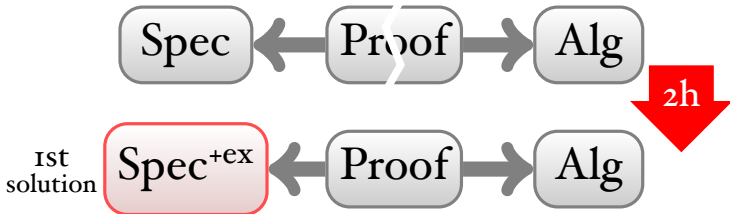
Idea:



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









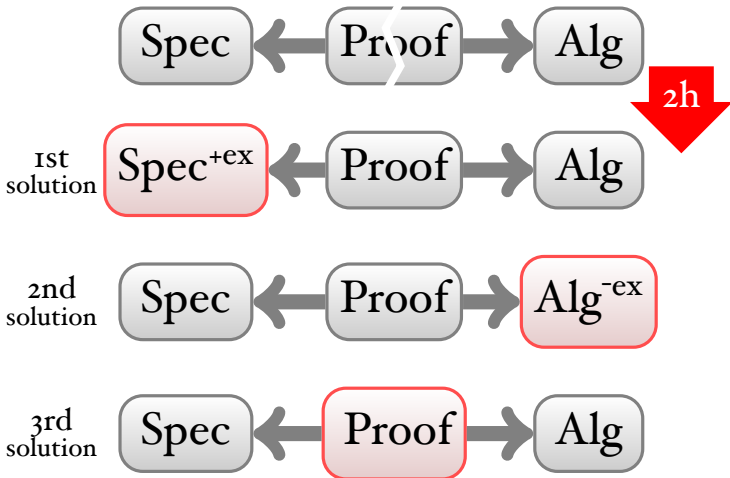
1st  
solution



2nd  
solution



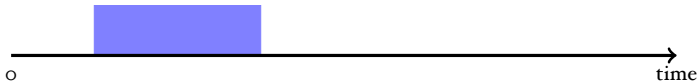




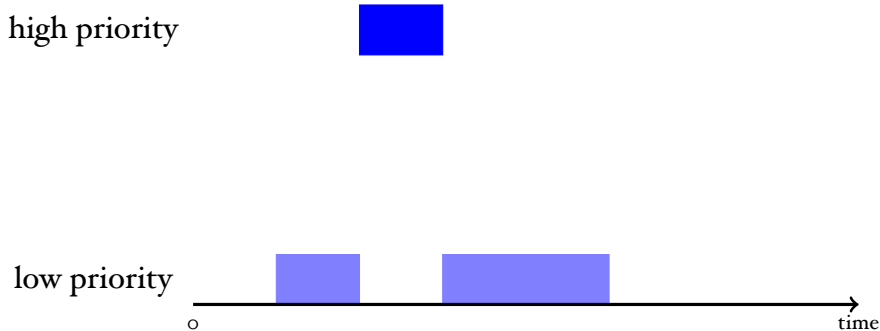
Each time one needs to check  $\sim 31$ pp of informal paper proofs. You have to be able to keep definitions and proofs consistent.

# Real-Time Scheduling

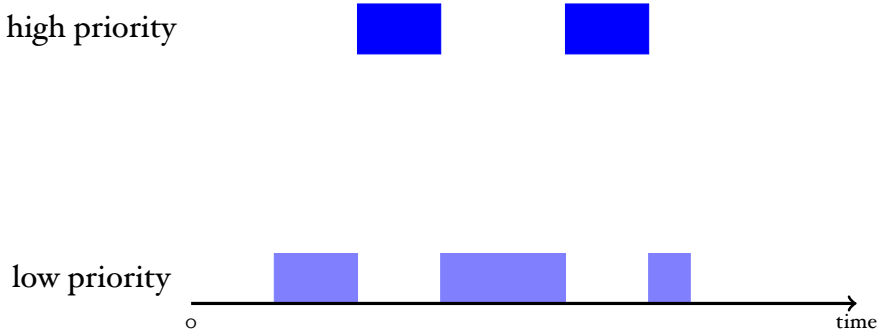
low priority



# Real-Time Scheduling



# Real-Time Scheduling



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

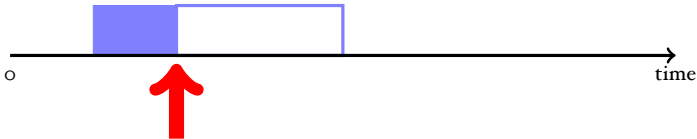
# Real-Time Scheduling

high priority



locks a resource

low priority



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

# Real-Time Scheduling

high priority



locks a resource

low priority



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

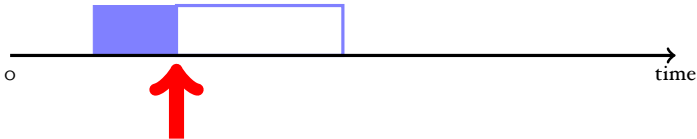
# Real-Time Scheduling

high priority



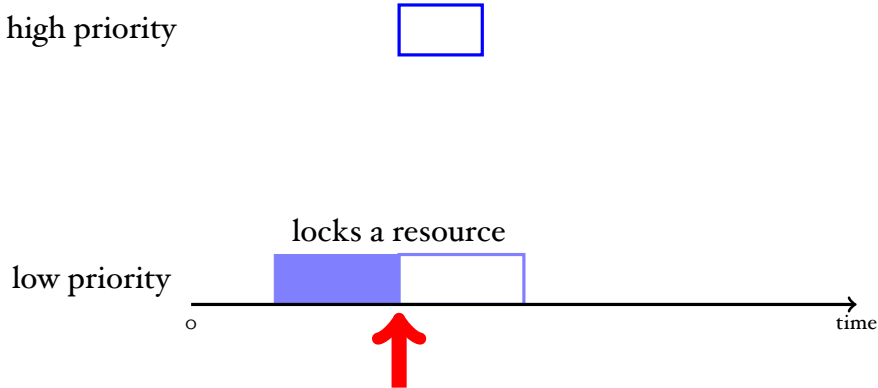
locks a resource

low priority



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

high priority

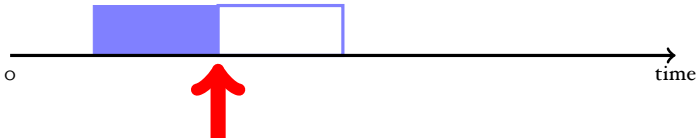


medium pr.



locks a resource

low priority



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

# Real-Time Scheduling

high priority

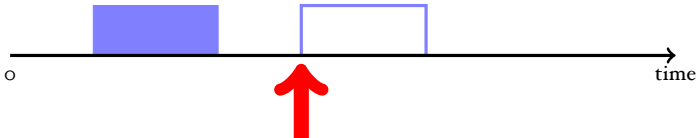


medium pr.



locks a resource

low priority



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

# Real-Time Scheduling

high priority

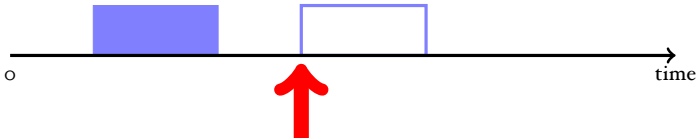


medium pr.



locks a resource

low priority



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

# Real-Time Scheduling

high priority

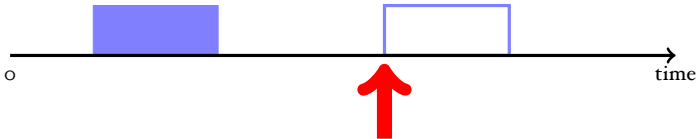


medium pr.



locks a resource

low priority



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

# Real-Time Scheduling

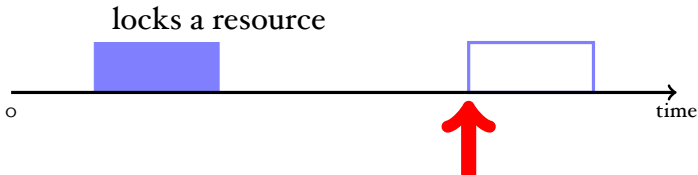
high priority



medium pr.



low priority



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

# Real-Time Scheduling

high priority



medium pr.

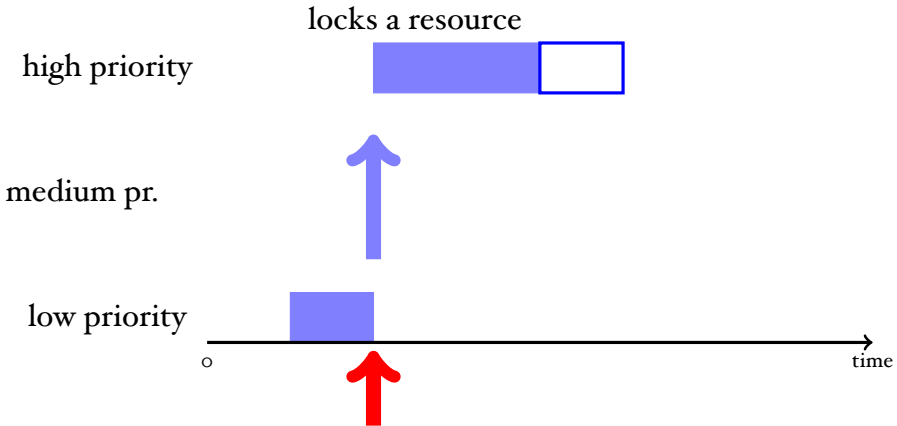
locks a resource

low priority



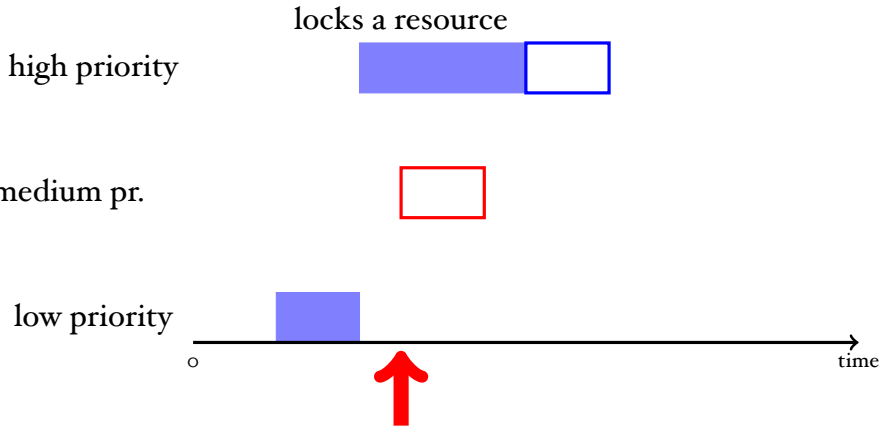
RT-Scheduling: You want to avoid that a high-priority process is starved indefinitely.

# 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

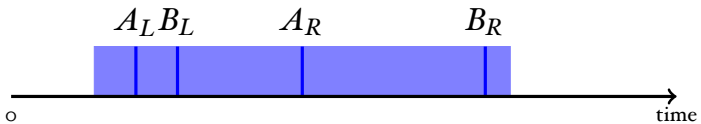
# Priority Inheritance Scheduling

- 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

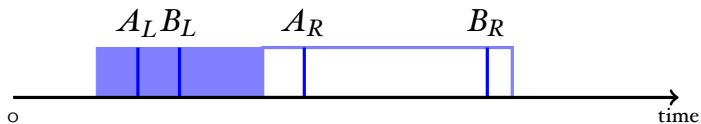
- Proved correct, reviewed in a respectable journal....what could possibly be wrong?

low priority



high priority

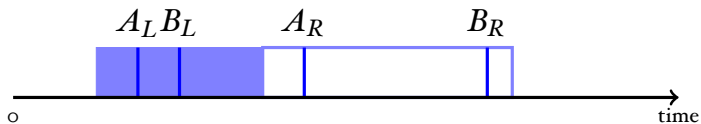
low priority



high priority



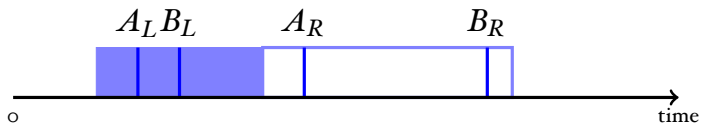
low priority



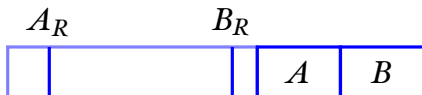
high priority



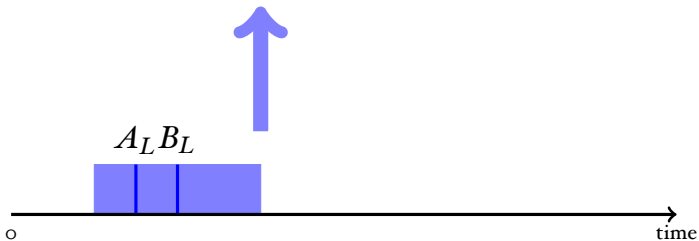
low priority



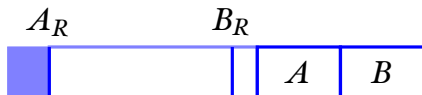
high priority



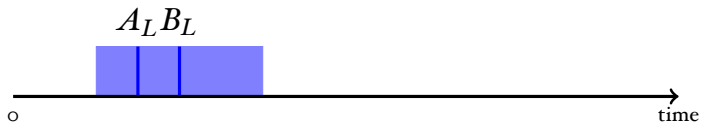
low priority



high priority



low priority

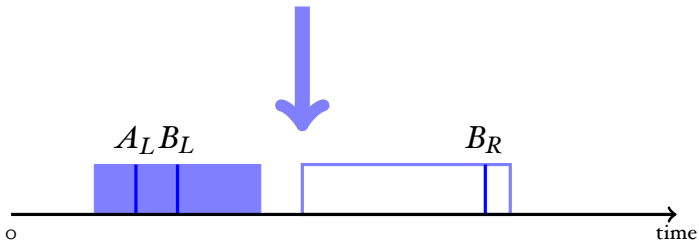




high priority



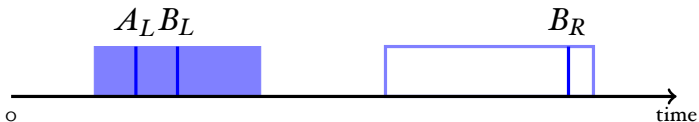
low priority



high priority



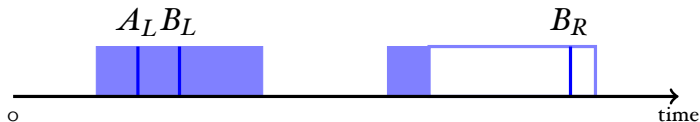
low priority



high priority



low priority



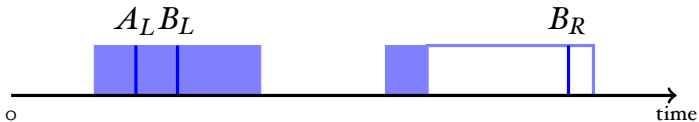
high priority

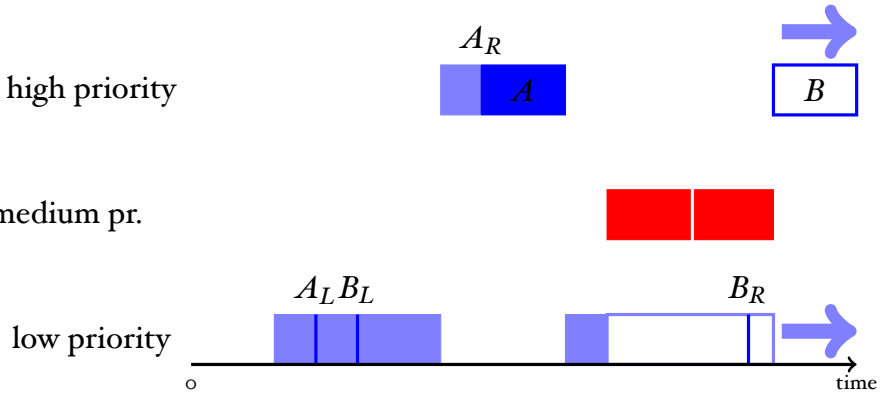


medium pr.



low priority





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

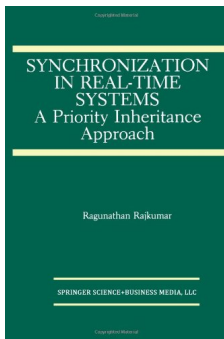
# Priority Inheritance Scheduling

- 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

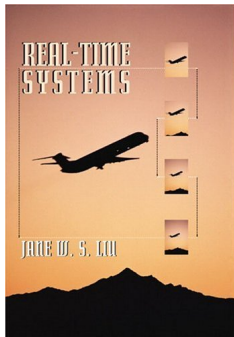
- Let a low priority process  $L$  temporarily inherit the high priority of  $H$  until  $L$  leaves the critical section unlocking the resource.
- Once the resource is unlocked,  $L$  returns to its original priority level. **BOGUS**
- ... $L$  needs to switch to the highest **remaining** priority of the threads that it blocks.

this error is already known since around 1999

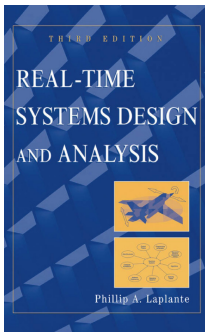


- by Rajkumar, 1991
- *“it resumes the priority it had at the point of entry into the critical section”*





- by Jane Liu, 2000
- *“The job  $J_1$  executes at its inherited priority until it releases  $R$ ; at that time, the priority of  $J_1$  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*”

GLOBAL  
EDITION



# Operating Systems

*Internals and Design Principles*

EIGHTH EDITION

William Stallings



# 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

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

# The Derivative of a Rexp

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

$der\ cr$  gives the answer, Brzozowski (1964), Owens (2005)  
“...have been lost in the sands of time...”



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

$$\text{nullable}(\emptyset) \stackrel{\text{def}}{=} \text{false}$$

$$\text{nullable}(\epsilon) \stackrel{\text{def}}{=} \text{true}$$

$$\text{nullable}(c) \stackrel{\text{def}}{=} \text{false}$$

$$\text{nullable}(r_1 + r_2) \stackrel{\text{def}}{=} \text{nullable}(r_1) \vee \text{nullable}(r_2)$$

$$\text{nullable}(r_1 \cdot r_2) \stackrel{\text{def}}{=} \text{nullable}(r_1) \wedge \text{nullable}(r_2)$$

$$\text{nullable}(r^*) \stackrel{\text{def}}{=} \text{true}$$

# The Derivative of a Rexp

$$\text{der } c (\emptyset) \stackrel{\text{def}}{=} \emptyset$$

$$\text{der } c (\epsilon) \stackrel{\text{def}}{=} \emptyset$$

$$\text{der } c (d) \stackrel{\text{def}}{=} \text{if } c = d \text{ then } \epsilon \text{ else } \emptyset$$

$$\text{der } c (r_1 + r_2) \stackrel{\text{def}}{=} \text{der } c r_1 + \text{der } c r_2$$

$$\text{der } c (r_1 \cdot r_2) \stackrel{\text{def}}{=} \text{if } \text{nullable}(r_1) \\ \text{then } (\text{der } c r_1) \cdot r_2 + \text{der } c r_2 \\ \text{else } (\text{der } c r_1) \cdot r_2$$

$$\text{der } c (r^*) \stackrel{\text{def}}{=} (\text{der } c r) \cdot (r^*)$$

# The Derivative of a Rexp

$$\text{der } c (\emptyset) \stackrel{\text{def}}{=} \emptyset$$

$$\text{der } c (\epsilon) \stackrel{\text{def}}{=} \emptyset$$

$$\text{der } c (d) \stackrel{\text{def}}{=} \text{if } c = d \text{ then } \epsilon \text{ else } \emptyset$$

$$\text{der } c (r_1 + r_2) \stackrel{\text{def}}{=} \text{der } c r_1 + \text{der } c r_2$$

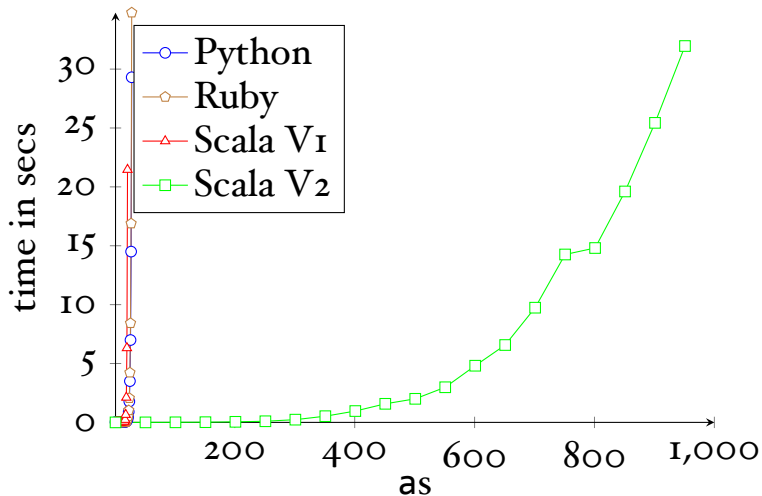
$$\text{der } c (r_1 \cdot r_2) \stackrel{\text{def}}{=} \text{if } \text{nullable}(r_1) \\ \text{then } (\text{der } c r_1) \cdot r_2 + \text{der } c r_2 \\ \text{else } (\text{der } c r_1) \cdot r_2$$

$$\text{der } c (r^*) \stackrel{\text{def}}{=} (\text{der } c r) \cdot (r^*)$$

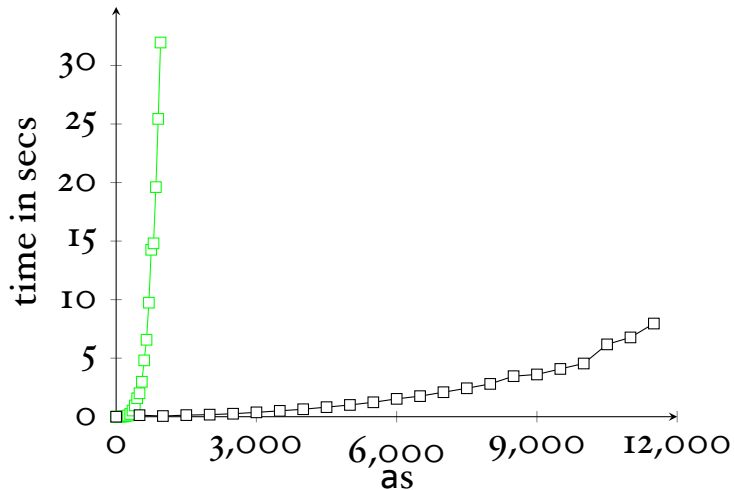
$$\text{ders } [] r \stackrel{\text{def}}{=} r$$

$$\text{ders } (c :: s) r \stackrel{\text{def}}{=} \text{ders } s (\text{der } c r)$$

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



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



# Correctness

It is a relative easy exercise in a theorem prover:

$matches(r, s)$  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](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:

$r ::= \emptyset$	$v ::=$
$\epsilon$	<i>Empty</i>
$c$	<i>Char</i> ( $c$ )
$r_1 \cdot r_2$	<i>Seq</i> ( $v_1, v_2$ )
$r_1 + r_2$	<i>Left</i> ( $v$ )
$r^*$	<i>Right</i> ( $v$ )
	$\square$
	$[v_1, \dots, v_n]$

# Regexes and Values

Regular expressions and their corresponding values:

$r ::= \emptyset$	$v ::=$
$\epsilon$	<i>Empty</i>
$c$	<i>Char</i> ( $c$ )
$r_1 \cdot r_2$	<i>Seq</i> ( $v_1, v_2$ )
$r_1 + r_2$	<i>Left</i> ( $v$ )
$r^*$	<i>Right</i> ( $v$ )
	$[]$
	$[v_1, \dots, v_n]$

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

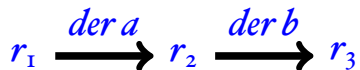
# Sulzmann & Lu Matcher

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

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

# Sulzmann & Lu Matcher

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



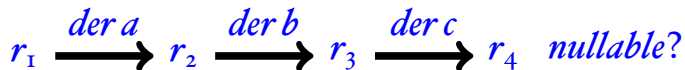
# Sulzmann & Lu Matcher

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



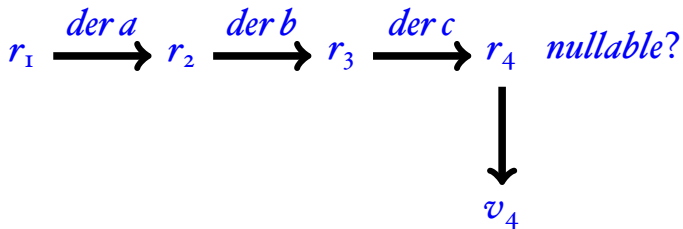
# Sulzmann & Lu Matcher

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



# Sulzmann & Lu Matcher

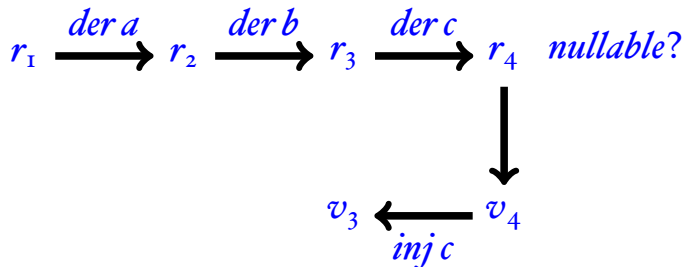
We want to match the string *abc* using  $r_1$ :





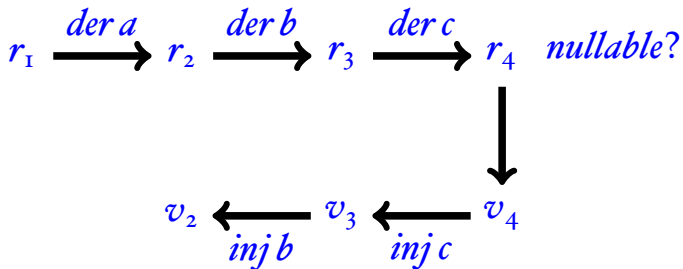
# Sulzmann & Lu Matcher

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



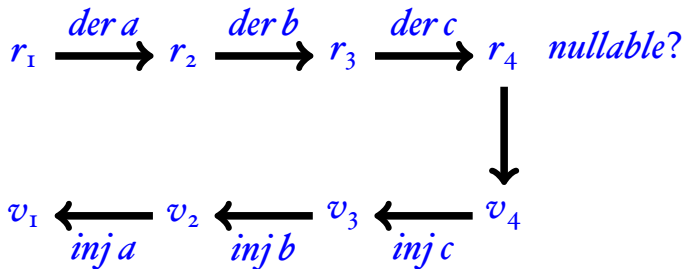
# Sulzmann & Lu Matcher

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



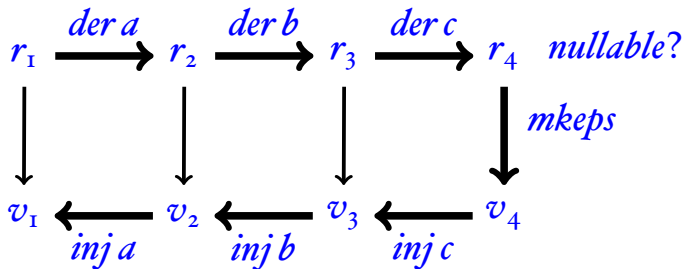
# Sulzmann & Lu Matcher

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.*

- If  $\vdash v : r$  and  $|v| = lw$  for some word  $w$ , then  $\vdash \text{proj}_{(r,l)} v : r \setminus l$ .*
- If  $\vdash v : r \setminus l$  then  $(\text{proj}_{(r,l)} \circ \text{inj}_{r \setminus l}) v = v$ .*
- If  $\vdash v : r$  and  $|v| = lw$  for some word  $w$ , then  $(\text{inj}_{r \setminus l} \circ \text{proj}_{(r,l)}) v = v$ .*

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

**MS:Possible fixes** We only ever apply  $\text{proj}$  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:  $[Right(Right(a, b))]$

$$\overline{\vdash \textit{Empty} : \epsilon}$$

$$\overline{\vdash \textit{Char}(c) : c}$$

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

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

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

$$\overline{\vdash [] : r^*}$$

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



# 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?

$$L(r) \neq \emptyset \Rightarrow \text{POSIX}(v, r)$$

# Problems

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

$$L(r) \neq \emptyset \Rightarrow \text{POSIX}(v, r)$$

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

$$|v_1| @ |v_2| = |v'_1| @ |v'_2|$$

# Problems

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

$$L(r) \neq \emptyset \Rightarrow \text{POSIX}(v, r)$$

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

$$|v_1| @ |v_2| = |v'_1| @ |v'_2|$$

- 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  $s \in r \rightarrow v$ :

$$\frac{}{[] \in \epsilon \rightarrow \textit{Empty}}$$

$$\frac{}{c \in c \rightarrow \textit{Char}(c)}$$

$$\frac{s \in r_1 \rightarrow v}{s \in r_1 + r_2 \rightarrow \textit{Left}(v)}$$

$$\frac{s \in r_2 \rightarrow v \quad s \notin L(r_1)}{s \in r_1 + r_2 \rightarrow \textit{Right}(v)}$$

$$s_1 \in r_1 \rightarrow v_1$$

$$s_2 \in r_2 \rightarrow v_2$$

$$\neg(\exists s_3 s_4. s_3 \neq [] \wedge s_3@s_4 = s_2 \wedge s_1@s_3 \in L(r_1) \wedge s_4 \in L(r_2))$$

$$\frac{}{s_1@s_2 \in r_1 \cdot r_2 \rightarrow \textit{Seq}(v_1, v_2)}$$

...



# Pencil-and-Paper Proofs in CS are normally incorrect

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

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

while

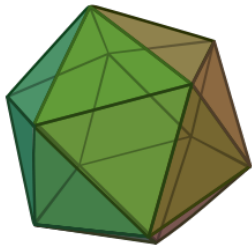
if  $\forall s. |v_2| \in L(\text{der } c \ r_1) \cdot s$  then  $\forall s. c \ |v_2| \in L(r_1) \cdot s$

is correct



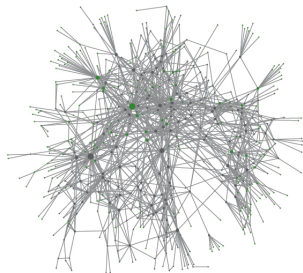
# Proofs in Math vs. in CS

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 applied 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?**