

POSIX Lexing with Derivatives of Regular Expressions

**Or, How to Find Bugs with the
Isabelle Theorem Prover**

Christian Urban

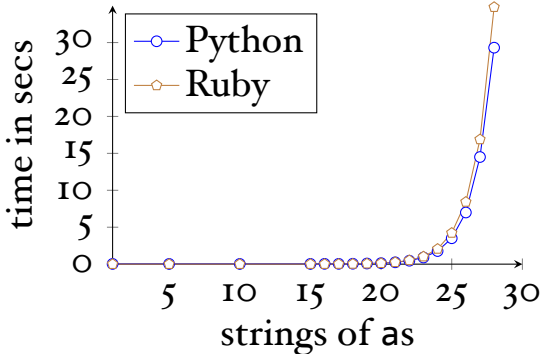
joint work with Fahad Ausaf and Roy Dyckhoff

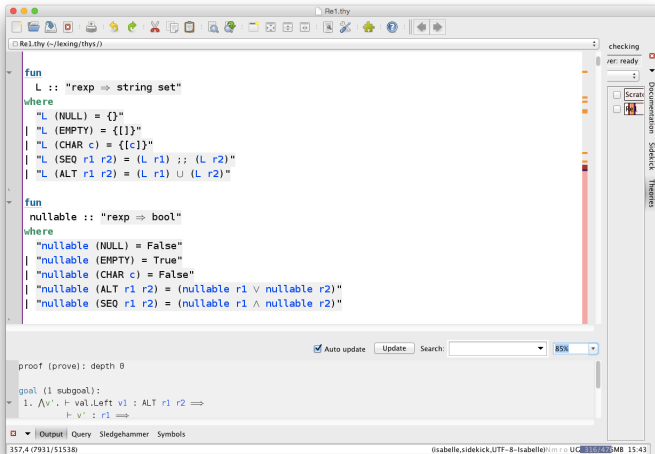
Why Bother?

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

Why Bother?

- Surely regular expressions must have been studied and implemented to death by now, no?
- ...well, take for example the “evil” regular expression $(a^?)^n \cdot a^n$ to match strings $\underbrace{a \dots a}_n$





- 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

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 (SN for a 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

$\lambda x. M$



$\forall x. P x$



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

$\lambda x. M$

$\forall x. P x$



- 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, \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**...?
- 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 on LF in
**ACM Transactions on
Computational Logic**,
2005, ~31pp

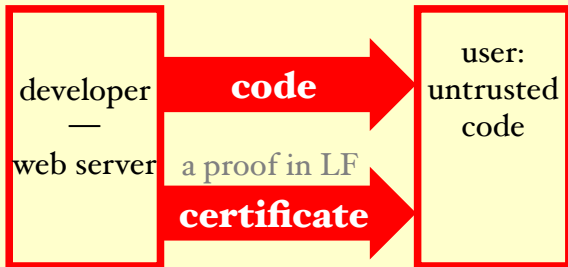


Andrew Appel

relied on their proof in a
security critical
application

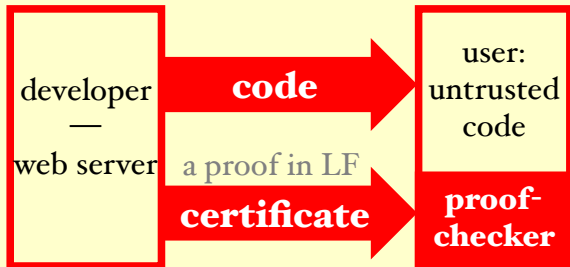
Proof-Carrying Code

Idea:



Proof-Carrying Code

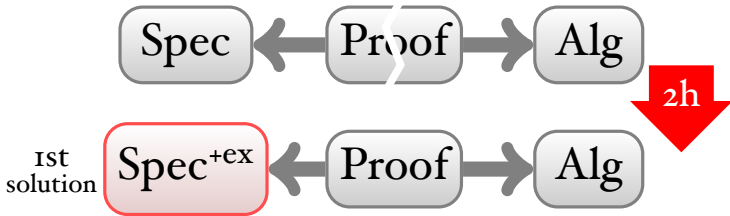
Idea:



- 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 (proved correct by Harper and Pfenning)







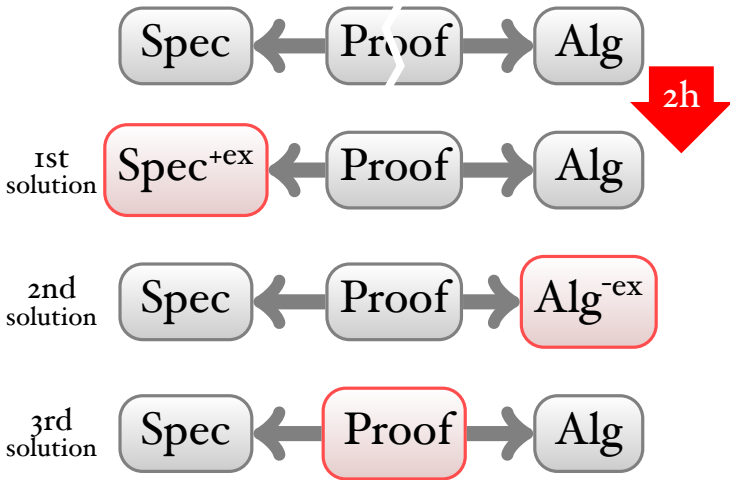


1st
solution



2nd
solution





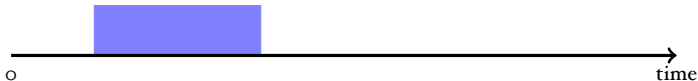
Each time one needs to check ~ 3 ipp of informal paper proofs—impossible without tool support. You have to be able to keep definitions and proofs consistent.

Lessons Learned

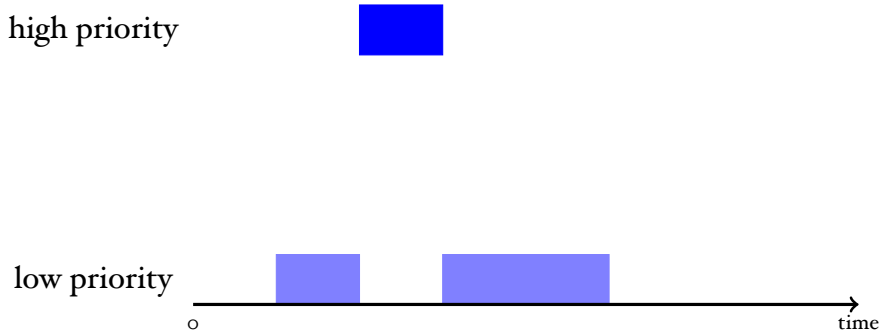
- by using a theorem prover we were able to keep a large proof consistent with changes in the first definitions
- it took us appr. 10 days to get to the error...probably the same time Harper and Pfenning needed to L^AT_EX their paper
- once there, we ran circles around them

Real-Time Scheduling

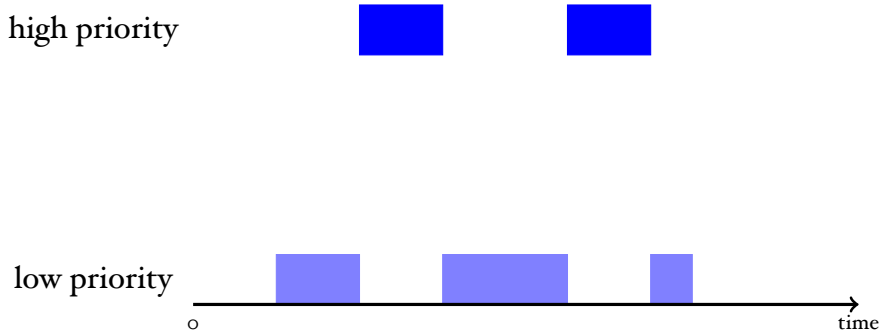
low priority



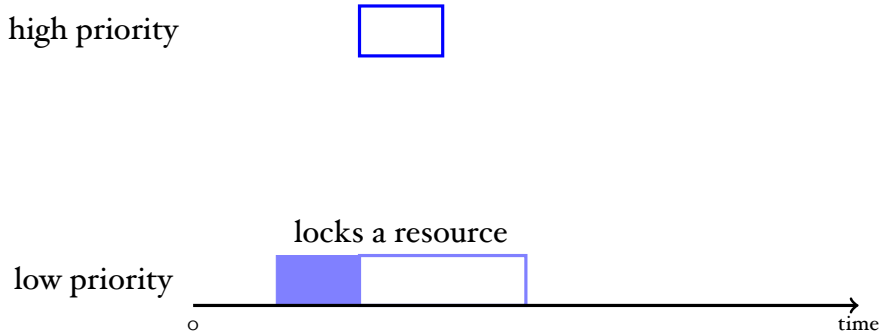
Real-Time Scheduling



Real-Time Scheduling



Real-Time Scheduling



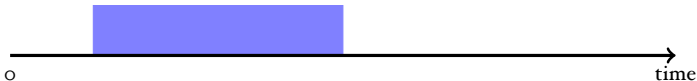
Real-Time Scheduling

high priority



locks a resource

low priority



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

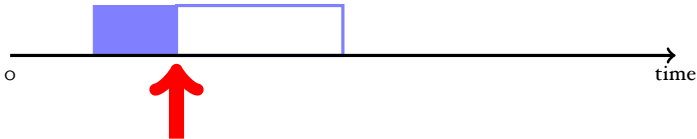
Real-Time Scheduling

high priority



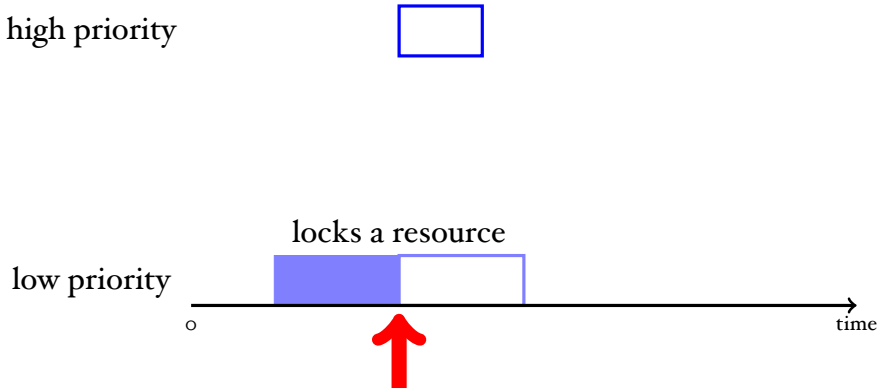
locks a resource

low priority



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

Real-Time Scheduling



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

Real-Time Scheduling

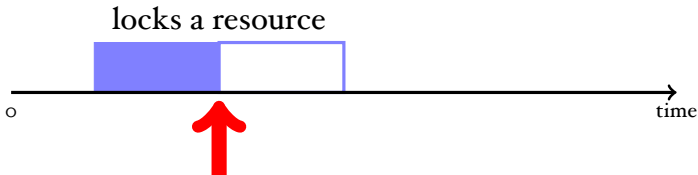
high priority



medium pr.

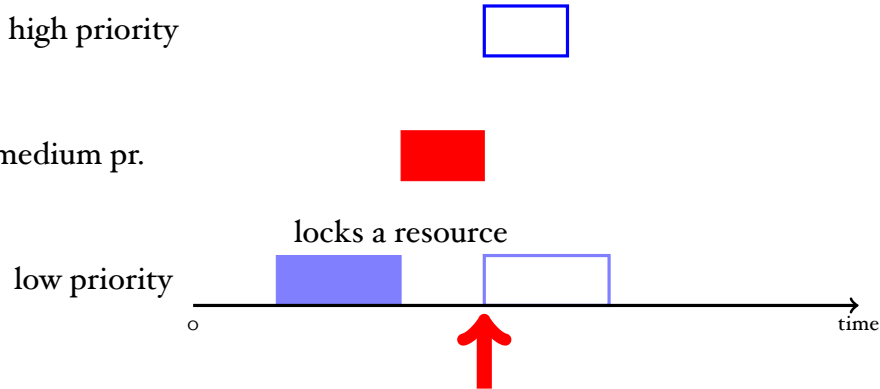


low priority



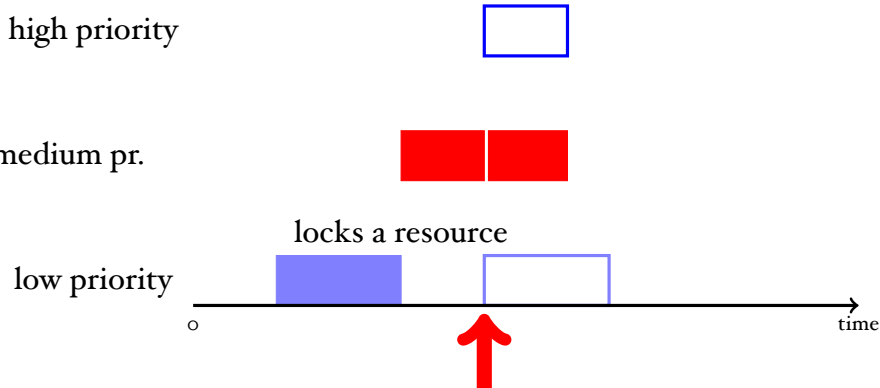
RT-Scheduling: You want to avoid that a high-priority process is starved indefinitely by lower priority processes.

Real-Time Scheduling



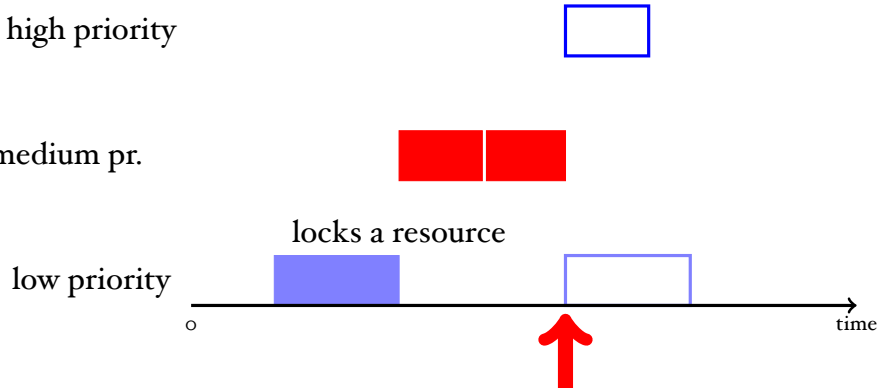
RT-Scheduling: You want to avoid that a high-priority process is starved indefinitely by lower priority processes.

Real-Time Scheduling



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

Real-Time Scheduling



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

Real-Time Scheduling

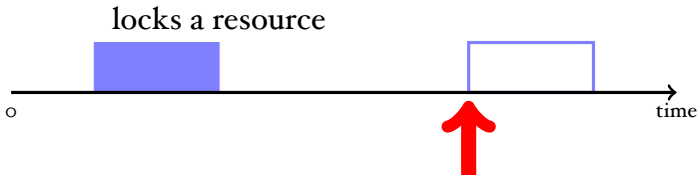
high priority



medium pr.



low priority



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

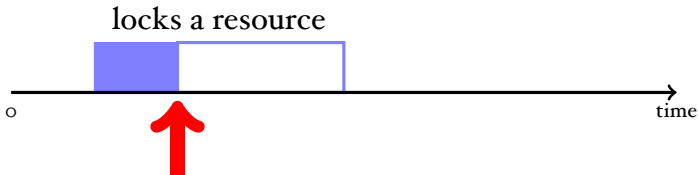
Real-Time Scheduling

high priority



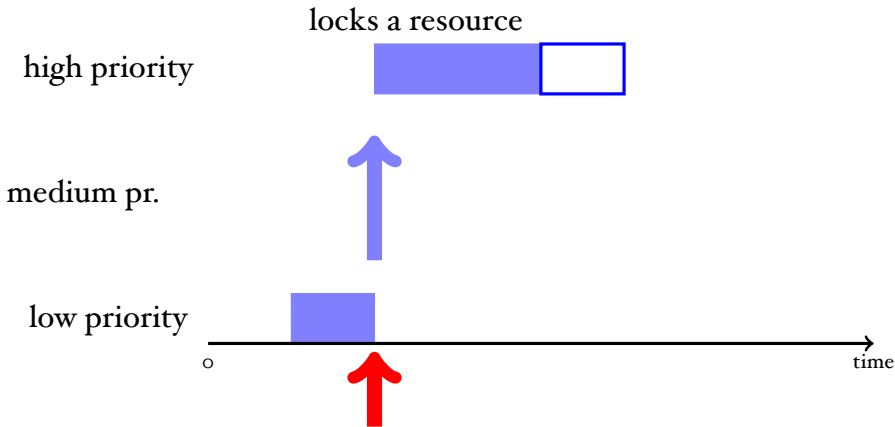
medium pr.

low priority



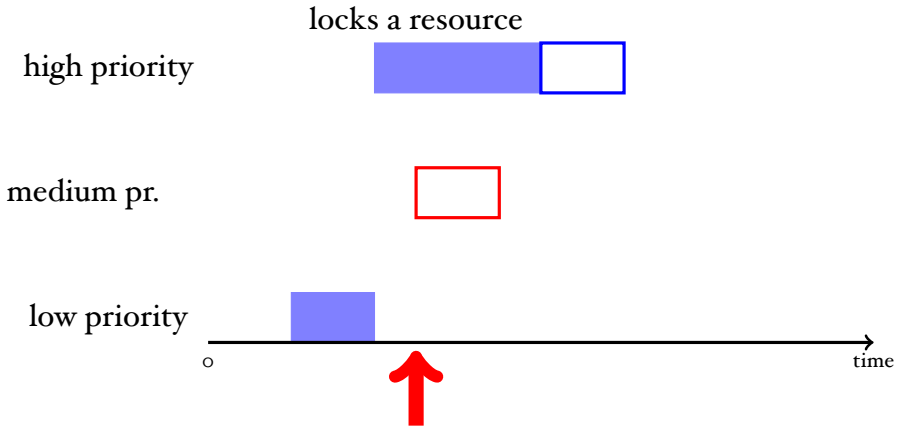
RT-Scheduling: You want to avoid that a high-priority process is starved indefinitely by lower priority processes.

Real-Time Scheduling



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

Real-Time Scheduling



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

Priority Inheritance Scheduling

- Idea: 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.”

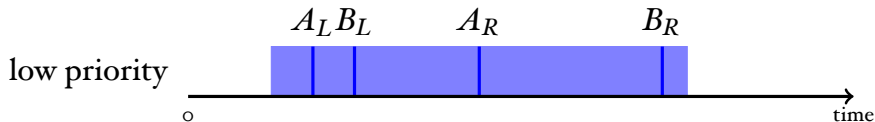
L. Sha, R. Rajkumar, and J. P. Lehoczky.
Priority Inheritance Protocols: An Approach to Real-Time Synchronization. IEEE Transactions on Computers, 39(9):1175–1185, 1990

Priority Inheritance Scheduling

- Idea: 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.”

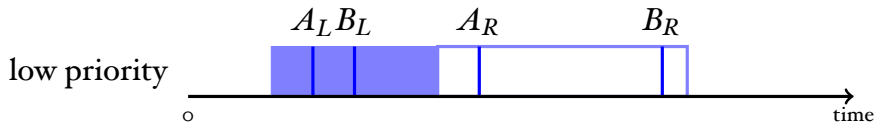
L. Sha, R. Rajkumar, and J. P. Lehoczky.
Priority Inheritance Protocols: An Approach to Real-Time Synchronization. IEEE Transactions on Computers, 39(9):1175–1185, 1990

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



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

high priority

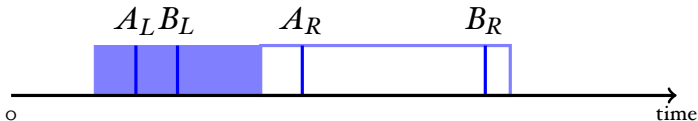


RT-Scheduling: You want to avoid that a high-priority process is starved indefinitely by lower priority processes.

high priority

A

low priority

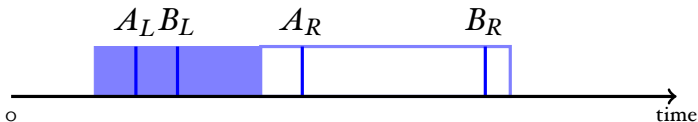


RT-Scheduling: You want to avoid that a high-priority process is starved indefinitely by lower priority processes.

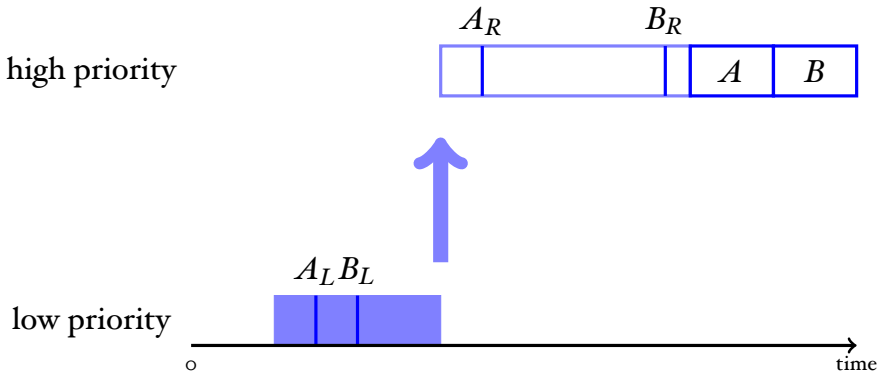
high priority



low priority

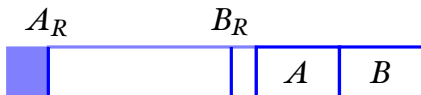


RT-Scheduling: You want to avoid that a high-priority process is starved indefinitely by lower priority processes.

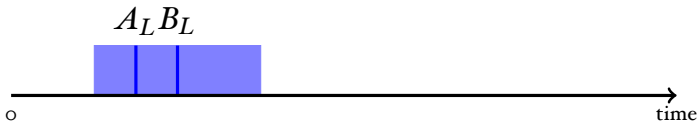


RT-Scheduling: You want to avoid that a high-priority process is starved indefinitely by lower priority processes.

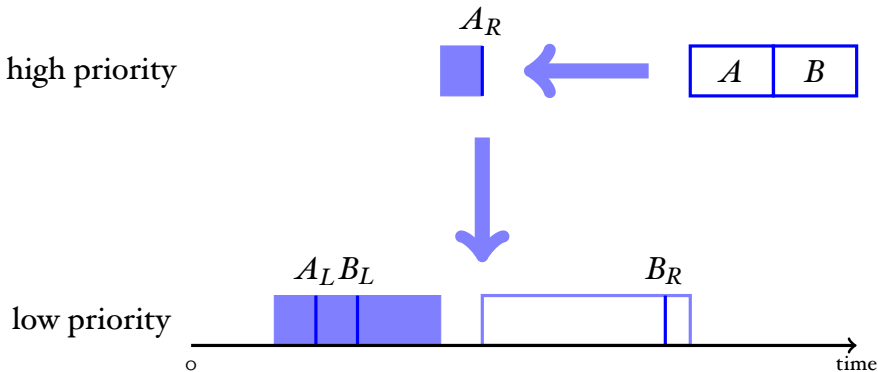
high priority



low priority

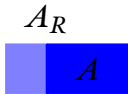


RT-Scheduling: You want to avoid that a high-priority process is starved indefinitely by lower priority processes.

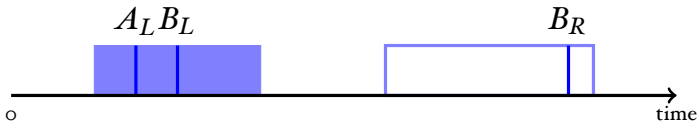


RT-Scheduling: You want to avoid that a high-priority process is starved indefinitely by lower priority processes.

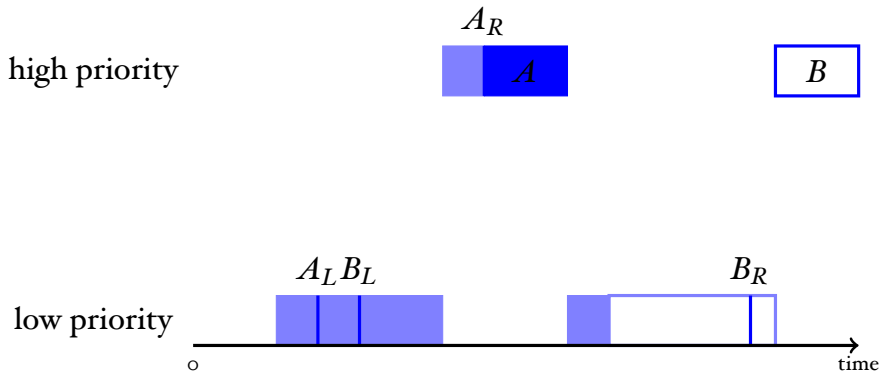
high priority



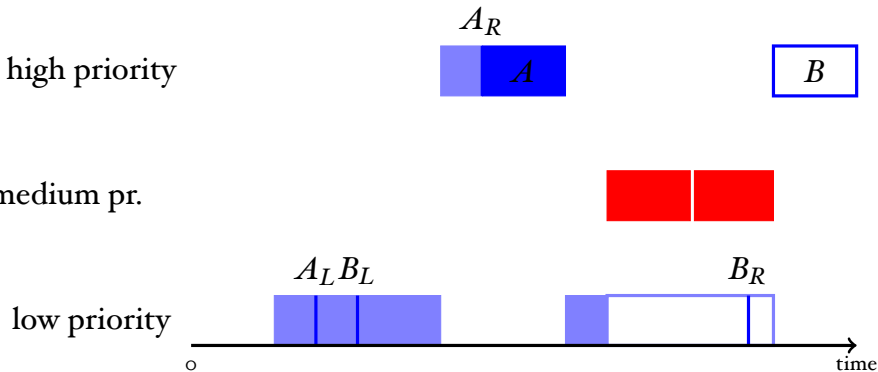
low priority



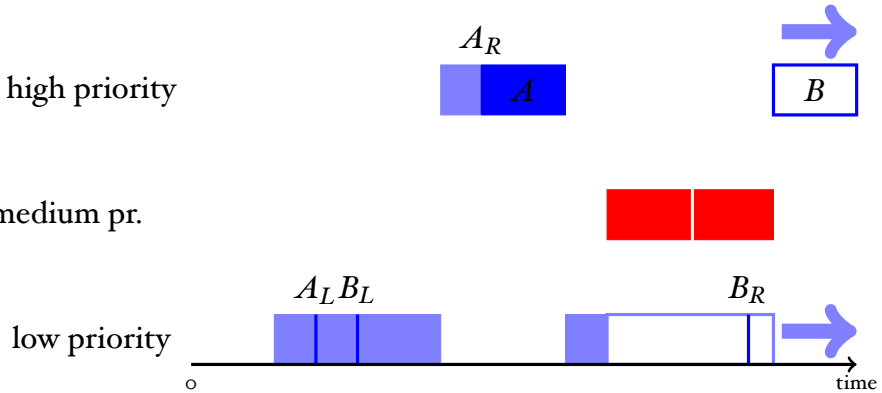
RT-Scheduling: You want to avoid that a high-priority process is starved indefinitely by lower priority processes.



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



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



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

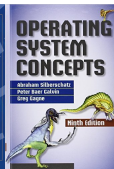
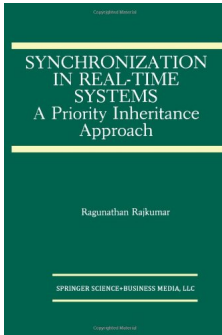
Priority Inheritance Scheduling

- Idea: 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**

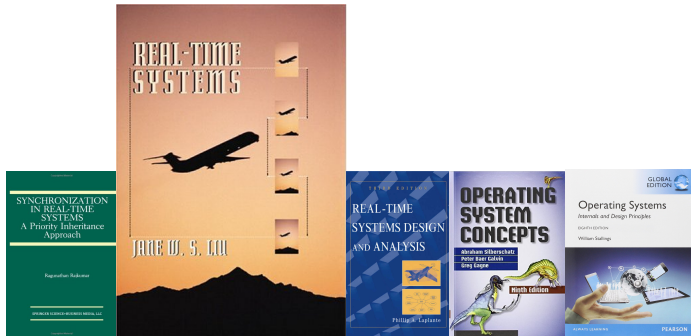
Priority Inheritance Scheduling

- Idea: 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.

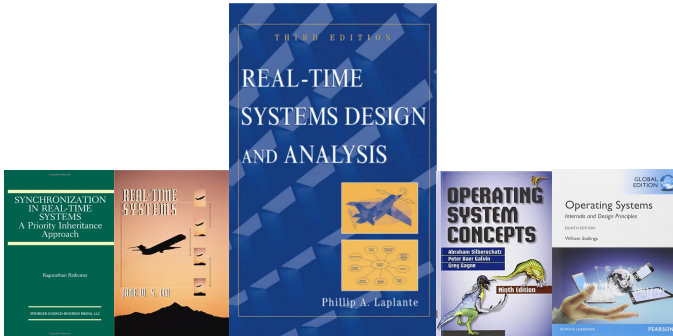
this error is already known since around 1999



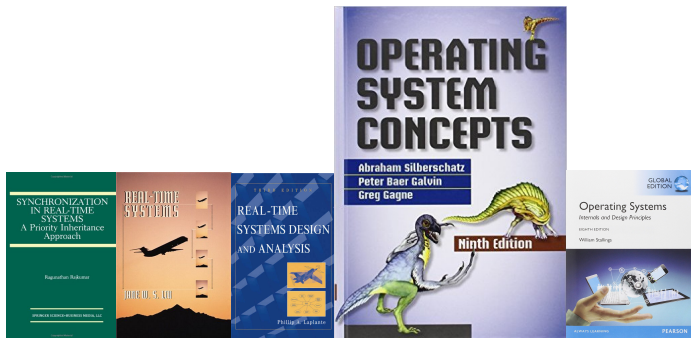
- by Rajkumar, 1991
- *“it resumes the priority it had at the point of entry into the critical section”*



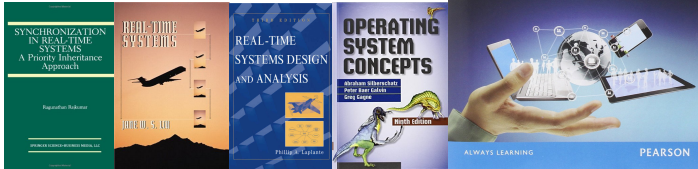
- by Jane Liu, 2000
- *“The job \mathcal{J}_1 executes at its inherited priority until it releases R ; at that time, the priority of \mathcal{J}_1 returns to its priority at the time when it acquires the resource R .”*
- gives pseudo code and uses pretty bogus data structures
- the interesting part is *“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.”*



- by Stallings (8th edition, 2014)
- *“This priority change takes place as soon as the higher-priority task blocks on the resource; it should end when the resource is released by the lower-priority task.”*

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
- we solved the single-processor case; the multi-processor case: no idea!

Regular Expressions

$r ::= \emptyset$	null
ϵ	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

$der\ c\ (\emptyset)$	$\stackrel{\text{def}}{=} \emptyset$
$der\ c\ (\epsilon)$	$\stackrel{\text{def}}{=} \emptyset$
$der\ c\ (d)$	$\stackrel{\text{def}}{=} \text{if } c = d \text{ then } \epsilon \text{ else } \emptyset$
$der\ c\ (r_1 + r_2)$	$\stackrel{\text{def}}{=} der\ c\ r_1 + der\ c\ r_2$
$der\ c\ (r_1 \cdot r_2)$	$\stackrel{\text{def}}{=} \text{if } \text{nullable}(r_1)$ then $(der\ c\ r_1) \cdot r_2 + der\ c\ r_2$ else $(der\ c\ r_1) \cdot r_2$
$der\ c\ (r^*)$	$\stackrel{\text{def}}{=} (der\ c\ r) \cdot (r^*)$

The Derivative of a Rexp

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

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

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

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

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

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

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

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

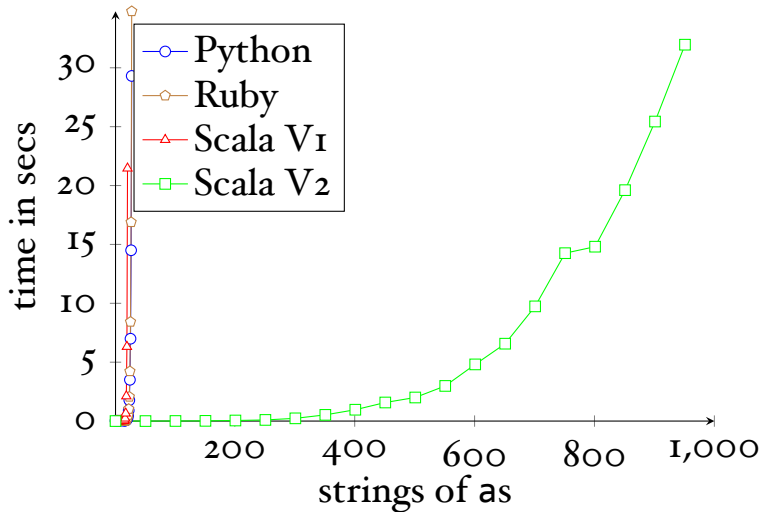
Correctness

It is a relative easy exercise in a theorem prover:

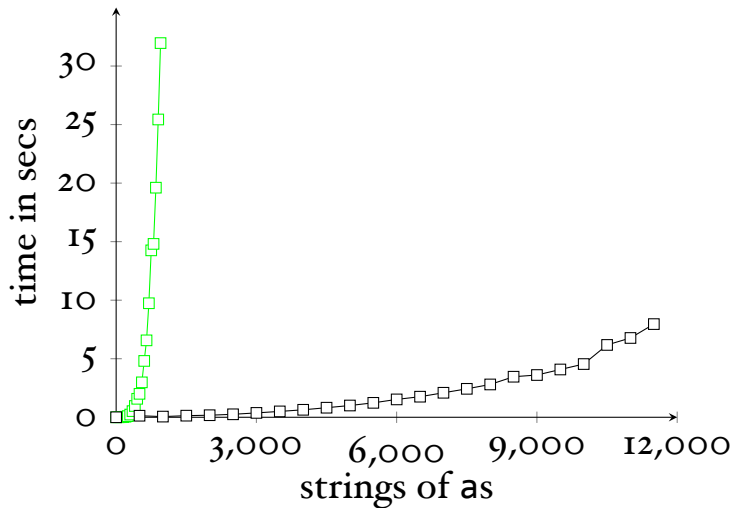
$matches(r, s)$ if and only if $s \in L(r)$

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

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



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



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 (for *how* a regular expression matched a string):

$r ::= \emptyset$	$v ::=$
ϵ	<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)
	$[]$
	$[v_1, \dots, v_n]$

Regexes and Values

Regular expressions and their corresponding values (for *how* a regular expression matched a string):

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

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

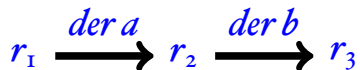
Sulzmann & Lu Matcher

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 :



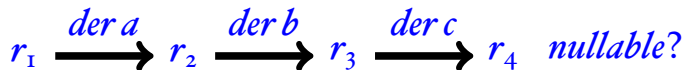
Sulzmann & Lu Matcher

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



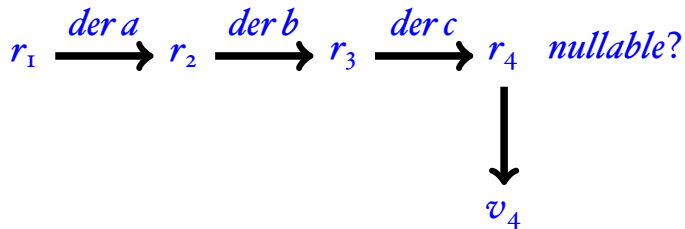
Sulzmann & Lu Matcher

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



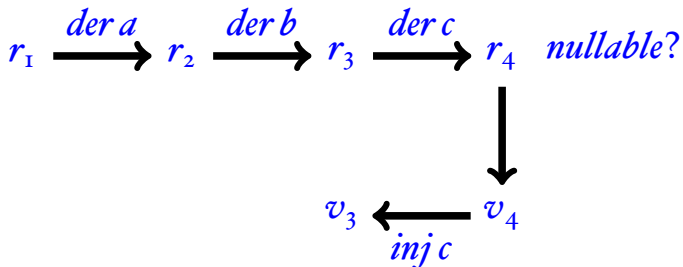
Sulzmann & Lu Matcher

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



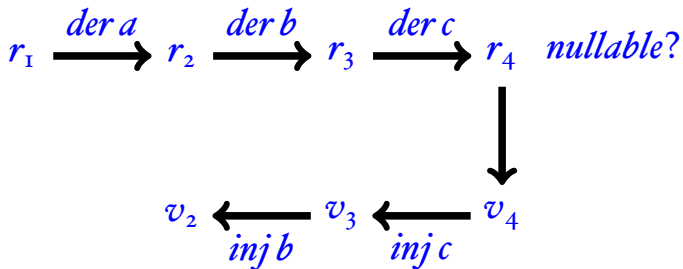
Sulzmann & Lu Matcher

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



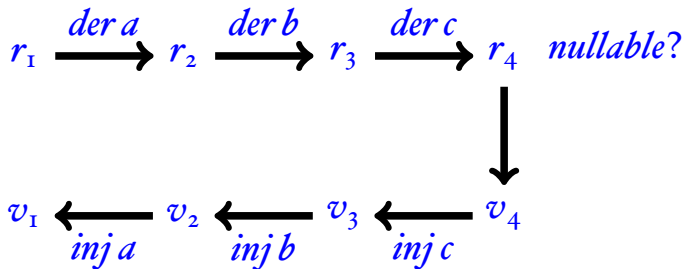
Sulzmann & Lu Matcher

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



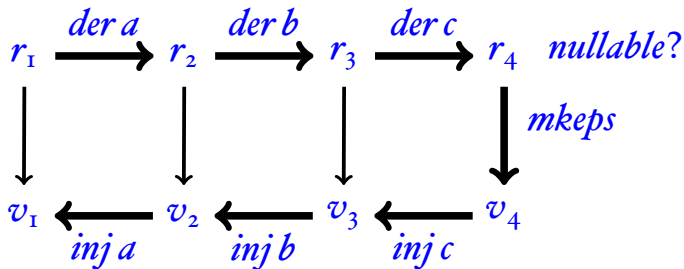
Sulzmann & Lu Matcher

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



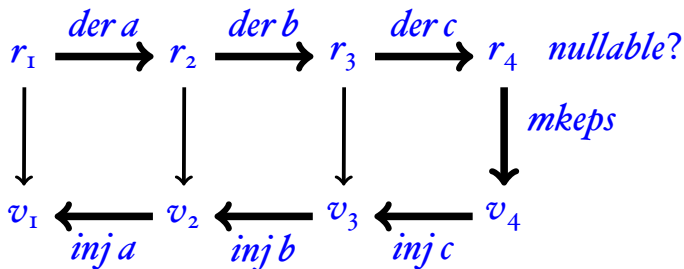
Sulzmann & Lu Matcher

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



Sulzmann & Lu Matcher

We want to match the string abc using r_1 :



The original ideas of Sulzmann and Lu are the *mkeps* and *inj* functions (omitted here).

Sulzmann & Lu Paper

- I have no doubt the algorithm is correct — the problem is 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 is 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 is 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 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: $[Right(Right(Seq(a, b)))]$

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

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

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

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

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

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

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

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?

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

Problems

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

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

- in the sequence case $\text{Seq}(v_1, v_2) \succ_{r_1 \cdot r_2} \text{Seq}(v'_1, v'_2)$, 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?

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

- in the sequence case $\text{Seq}(v_1, v_2) \succ_{r_1 \cdot r_2} \text{Seq}(v'_1, v'_2)$, 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

- a direct definition of what a POSIX value is, using the relation $s \in r \rightarrow v$ (specification):

$$\overline{\square} \in \epsilon \rightarrow \text{Empty}$$

$$\overline{c} \in c \rightarrow \text{Char}(c)$$

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

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

$$s_1 \in r_1 \rightarrow v_1$$

$$s_2 \in r_2 \rightarrow v_2$$

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

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

...

Properties

It is almost trivial to prove:

- Uniqueness

If $s \in r \rightarrow v_1$ and $s \in r \rightarrow v_2$ then $v_1 = v_2$.

- Correctness

$lexer(r, s) = v$ if and only if $s \in r \rightarrow v$

Properties

It is almost trivial to prove:

- Uniqueness

If $s \in r \rightarrow v_1$ and $s \in r \rightarrow v_2$ then $v_1 = v_2$.

- Correctness

$lexer(r, s) = v$ if and only if $s \in r \rightarrow v$

You can now start to implement optimisations and derive correctness proofs for them. But we still do not know whether

$$s \in r \rightarrow v$$

is a POSIX value according to Sulzmann & Lu's definition (biggest value for s and r)

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

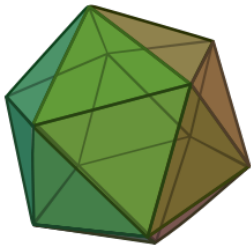
if $\forall s. |v_2| \in L(\text{der } c r_1) \cdot s$ then $\forall s. c |v_2| \in L(r_1) \cdot s$

is correct



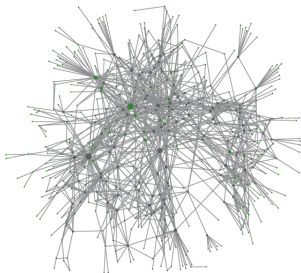
Proofs in Math vs. in CS

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 replaced the POSIX definition of Sulzmann & Lu by a new definition (ours is inspired by work of Vansummeren, 2006)
- their proof contained small gaps (acknowledged) but had also fundamental flaws
- now, 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, is a small beautiful functional program

Questions?