Error-Free Programmin with Theorem Provers

Christian Urban

Technical University of Munich, Germany

in Nanjing on the kind invitation of
Professor Xingyuan Zhang and his group

My Background

- researcher in Theoretical Computer Science
- programmer on a software system with 800 kloc (though I am responsible only for 35 kloc)

My Background

- researcher in Theoretical Computer Science
- programmer on a software system with 800 kloc (though I am responsible only for 35 kloc)

> A theorem prover called **Isabelle**.

My Background

- researcher in Theoretical Computer Science
- programmer on a software system with 800 kloc (though I am responsible only for 35 kloc)

A theorem prover called **Isabelle**.

Like every other code, this code is very hard to get correct.

Regular Expressions

An example many (should) know about:

Regular Expressions:

Regular Expressions

An example many (should) know about:
Regular Expressions:

$$[] \quad | \quad c \quad | \quad r_1|r_2 \quad | \quad r_1 \cdot r_2 \quad | \quad r^*$$

$(a \cdot b)^* \quad \mapsto \quad \{[], ab, abab, ababab, ...\}$
$x \cdot (0 \mid 1 \mid 2 \ldots 8 \mid 9)^* \quad \mapsto \quad \{x, x0, x1, ..., x00, ..., x123, ...\}$

Regular Expressions

An example many (should) know about:
Regular Expressions:

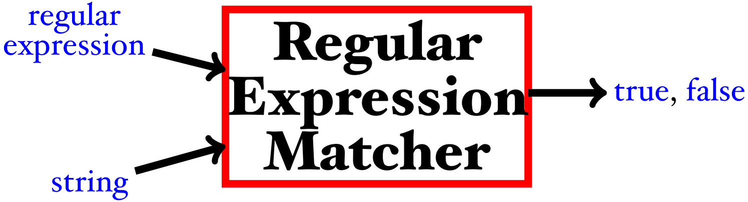
r	$::=$	NULL	(matches no string)
	$\mid$	EMPTY	(matches the empty string, [])
	$\mid$	CHR c	(matches the character c)
	$\mid$	ALT r_1 r_2	(alternative, $r_1 \mid r_2$)
	$\mid$	SEQ r_1 r_2	(sequential, $r_1 \cdot r_2$)
	$\mid$	STAR r	(repeat, r^*)

$(a \cdot b)^* \mapsto \{[], ab, abab, ababab, \ldots\}$

$x \cdot (0 \mid 1 \mid 2 \ldots 8 \mid 9)^* \mapsto \{x, x0, x1, \ldots, x00, \ldots, x123, \ldots\}$

RegExp Matcher

Let's implement a regular expression matcher:



RegExp Matcher

input: a list of RegExps and a string **output:** true or false

RegExp Matcher

input: a <u>list</u> of RegExps and a string **output:** true or false

$$
\begin{array}{lcl}
\text{match [] []} & = & \text{true} \\
\text{match [] \_} & = & \text{false} \\
\text{match (NULL::rs) s} & = & \text{false} \\
\text{match (EMPTY::rs) s} & = & \text{match rs s} \\
\text{match (CHR c::rs) (c::s)} & = & \text{match rs s} \\
\text{match (CHR c::rs) \_} & = & \text{false} \\
\text{match (ALT } r_1\ r_2\text{::rs) s} & = & \text{match (}r_1\text{::rs) s} \\
 & & \quad \text{orelse match (}r_2\text{::rs) s} \\
\text{match (SEQ } r_1\ r_2\text{::rs) s} & = & \text{match (}r_1\text{::}r_2\text{::rs) s} \\
\text{match (STAR r::rs) s} & = & \text{match rs s} \\
 & & \quad \text{orelse match (r::STAR r::rs) s}
\end{array}
$$

RegExp Matcher

input: a <u>list</u> of RegExps and a string **output:** true or false

$$
\begin{array}{lcl}
\text{match [] []} & = & \text{true} \\
\text{match [] \_} & = & \text{false} \\
\text{match (NULL::rs) s} & = & \text{false} \\
\text{match (EMPTY::rs) s} & = & \text{match rs s} \\
\text{match (CHR c::rs) (c::s)} & = & \text{match rs s} \\
\text{match (CHR c::rs) \_} & = & \text{false} \\
\text{match (ALT } r_1\ r_2\text{::rs) s} & = & \text{match (}r_1\text{::rs) s} \\
 & & \quad \text{orelse match (}r_2\text{::rs) s} \\
\text{match (SEQ } r_1\ r_2\text{::rs) s} & = & \text{match (}r_1\text{::}r_2\text{::rs) s} \\
\text{match (STAR r::rs) s} & = & \text{match rs s} \\
 & & \quad \text{orelse match (r::STAR r::rs) s}
\end{array}
$$

we start the program with

matches r s = match [r] s

Program in Scala

```
sealed abstract class Rexp
sealed case class Null extends Rexp
sealed case class Empty extends Rexp
sealed case class Chr(c: Char) extends Rexp
sealed case class Alt(r1 : Rexp, r2 : Rexp) extends Rexp
sealed case class Seq(r1 : Rexp, r2 : Rexp) extends Rexp
sealed case class Star(r : Rexp) extends Rexp

def match1 (rs : List[Rexp], s : List[Char]) : Boolean = rs match {
  case Nil ⇒ if (s == Nil) true else false
  case (Null()::rs) ⇒ false
  case (Empty()::rs) ⇒ match1 (rs, s)
  case (Chr(c)::rs) ⇒ s match
    { case Nil ⇒ false
      case (d::s) ⇒ if (c==d) match1 (rs,s) else false }
  case (Alt (r1, r2)::rs) ⇒ match1 (r1::rs, s) || match1 (r2::rs, s)
  case (Seq (r1, r2)::rs) ⇒ match1 (r1::r2::rs, s)
  case (Star (r)::rs) ⇒ match1 (r::rs, s) || match1 (r::Star (r)::rs, s)
}
```

Testing

Every good programmer should do thourough tests:

matches (a·b)*	[]	$\mapsto$	true
matches (a·b)*	ab	$\mapsto$	true
matches (a·b)*	aba	$\mapsto$	false
matches (a·b)*	abab	$\mapsto$	true
matches (a·b)*	abaa	$\mapsto$	false

Testing

Every good programmer should do thourough tests:

matches $(a \cdot b)^*$	[]	$\mapsto$	true
matches $(a \cdot b)^*$	ab	$\mapsto$	true
matches $(a \cdot b)^*$	aba	$\mapsto$	false
matches $(a \cdot b)^*$	abab	$\mapsto$	true
matches $(a \cdot b)^*$	abaa	$\mapsto$	false
matches $x \cdot (0 \mid 1)^*$	x	$\mapsto$	true
matches $x \cdot (0 \mid 1)^*$	x0	$\mapsto$	true
matches $x \cdot (0 \mid 1)^*$	x3	$\mapsto$	false

Testing

Every good programmer should do thourough tests:

matches $(a \cdot b)^*$	[]	$\mapsto$	true
matches $(a \cdot b)^*$	ab	$\mapsto$	true
matches $(a \cdot b)^*$	aba	$\mapsto$	false
matches $(a \cdot b)^*$	abab	$\mapsto$	true
matches $(a \cdot b)^*$	abaa	$\mapsto$	false
matches $x \cdot (0 \mid 1)^*$	x	$\mapsto$	true
matches $x \cdot (0 \mid 1)^*$	x0	$\mapsto$	true
matches $x \cdot (0 \mid 1)^*$	x3	$\mapsto$	false

looks OK ...let's ship it to customers

Testing

- While testing is an important part in the process of programming development

Testing

- While testing is an important part in the process of programming development
- we can only test a **finite** amount of examples

Testing

- While testing is an important part in the process of programming development
- we can only test a **finite** amount of examples

> "Testing can only show the presence of errors, never their absence" (Edsger W. Dijkstra)

Testing

- While testing is an important part in the process of programming development
- we can only test a **finite** amount of examples

"Testing can only show the presence of errors, never their absence" (Edsger W. Dijkstra)

- In a theorem prover we can establish properties that apply to **all** input and **all** output.

Testing

- While testing is an important part in the process of programming development
- we can only test a **finite** amount of examples

> "Testing can only show the presence of errors, never their absence" (Edsger W. Dijkstra)

- In a theorem prover we can establish properties that apply to **all** input and **all** output.
- For example we can establish that the matcher terminates on all input.

RegExp Matcher

We need to find a measure that gets smaller in each recursive call.

needs to get smaller →

$$
\begin{array}{lcl}
\text{match [] []} & = & \text{true} \\
\text{match [] \_} & = & \text{false} \\
\text{match (NULL::rs) s} & = & \text{false} \\
\text{match (EMPTY::rs) s} & = & \text{match rs s} \\
\text{match (CHR c::rs) (c::s)} & = & \text{match rs s} \\
\text{match (CHR c::rs) \_} & = & \text{false} \\
\text{match (ALT } r_1\ r_2\text{::rs) s} & = & \text{match (}r_1\text{::rs) s} \\
 & & \quad \text{orelse match (}r_2\text{::rs) s} \\
\text{match (SEQ } r_1\ r_2\text{::rs) s} & = & \text{match (}r_1\text{::}r_2\text{::rs) s} \\
\text{match (STAR r::rs) s} & = & \text{match rs s} \\
 & & \quad \text{orelse match (r::STAR r::rs) s}
\end{array}
$$

RegExp Matcher

We need to find a measure that gets smaller in each recursive call.

needs to get smaller →

match [] []	=	true	✓
match [] _	=	false	✓
match (NULL::rs) s	=	false	✓
match (EMPTY::rs) s	=	match rs s	
match (CHR c::rs) (c::s)	=	match rs s	
match (CHR c::rs) _	=	false	✓
match (ALT r_1 r_2::rs) s	=	match (r_1::rs) s orelse match (r_2::rs) s	
match (SEQ r_1 r_2::rs) s	=	match (r_1::r_2::rs) s	
match (STAR r::rs) s	=	match rs s orelse match (r::STAR r::rs) s	

RegExp Matcher

We need to find a measure that gets smaller in each recursive call.

needs to get smaller →

match [] []	=	true	✓
match [] _	=	false	✓
match (NULL::rs) s	=	false	✓
match (EMPTY::rs) s	=	match rs s	✓
match (CHR c::rs) (c::s)	=	match rs s	
match (CHR c::rs) _	=	false	✓
match (ALT r_1 r_2::rs) s	=	match (r_1::rs) s orelse match (r_2::rs) s	
match (SEQ r_1 r_2::rs) s	=	match (r_1::r_2::rs) s	
match (STAR r::rs) s	=	match rs s orelse match (r::STAR r::rs) s	

RegExp Matcher

We need to find a measure that gets smaller in each recursive call.

needs to get smaller →

match [] []	=	true	✓
match [] _	=	false	✓
match (NULL::rs) s	=	false	✓
match (EMPTY::rs) s	=	match rs s	✓
match (CHR c::rs) (c::s)	=	match rs s	✓
match (CHR c::rs) _	=	false	✓
match (ALT r_1 r_2::rs) s	=	match (r_1::rs) s orelse match (r_2::rs) s	
match (SEQ r_1 r_2::rs) s	=	match (r_1::r_2::rs) s	
match (STAR r::rs) s	=	match rs s orelse match (r::STAR r::rs) s	

RegExp Matcher

We need to find a measure that gets smaller in each recursive call.

needs to get smaller →

match [] []	=	true	✓
match [] _	=	false	✓
match (NULL::rs) s	=	false	✓
match (EMPTY::rs) s	=	match rs s	✓
match (CHR c::rs) (c::s)	=	match rs s	✓
match (CHR c::rs) _	=	false	✓
match (ALT r_1 r_2::rs) s	=	match (r_1::rs) s orelse match (r_2::rs) s	✓
match (SEQ r_1 r_2::rs) s	=	match (r_1::r_2::rs) s	
match (STAR r::rs) s	=	match rs s orelse match (r::STAR r::rs) s	

RegExp Matcher

We need to find a measure that gets smaller in each recursive call.

needs to get smaller →

match [] []	=	true	✓
match [] _	=	false	✓
match (NULL::rs) s	=	false	✓
match (EMPTY::rs) s	=	match rs s	✓
match (CHR c::rs) (c::s)	=	match rs s	✓
match (CHR c::rs) _	=	false	✓
match (ALT r_1 r_2::rs) s	=	match (r_1::rs) s orelse match (r_2::rs) s	✓
match (SEQ r_1 r_2::rs) s	=	match (r_1::r_2::rs) s	✓
match (STAR r::rs) s	=	match rs s orelse match (r::STAR r::rs) s	

RegExp Matcher

We need to find a measure that gets smaller in each recursive call.

needs to get smaller →

match [] []	=	true	✓
match [] _	=	false	✓
match (NULL::rs) s	=	false	✓
match (EMPTY::rs) s	=	match rs s	✓
match (CHR c::rs) (c::s)	=	match rs s	✓
match (CHR c::rs) _	=	false	✓
match (ALT r_1 r_2::rs) s	=	match (r_1::rs) s orelse match (r_2::rs) s	✓
match (SEQ r_1 r_2::rs) s	=	match (r_1::r_2::rs) s	✓
match (STAR r::rs) s	=	match rs s orelse match (r::STAR r::rs) s	✗

Bug Hunting

Several hours later...

Bug Hunting

matches (STAR (EMPTY)) s $\mapsto$ loops

Bug Hunting

$$\text{matches (STAR (EMPTY)) s} \quad \mapsto \quad \text{loops}$$

$$\begin{aligned}
&\ldots\\
&\text{match (EMPTY::rs) s} \;=\; \text{match rs s}\\
&\ldots\\
&\text{match (STAR r::rs) s} \;=\; \text{match rs s}\\
&\qquad\qquad\qquad\qquad\quad \text{orelse match (r::STAR r::rs) s}
\end{aligned}$$

Bug Hunting

$$
\begin{array}{lcl}
\text{matches (STAR (EMPTY)) s} & \mapsto & \text{loops} \\
\text{matches (STAR (EMPTY | ...)) s} & \mapsto & \text{loops}
\end{array}
$$

...

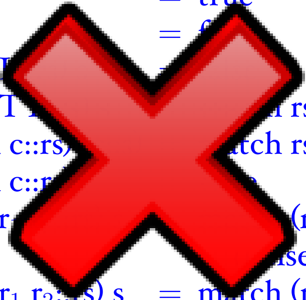
$$
\begin{array}{lcl}
\text{match (EMPTY::rs) s} & = & \text{match rs s}
\end{array}
$$

...

$$
\begin{array}{lcl}
\text{match (STAR r::rs) s} & = & \text{match rs s} \\
 & & \quad \text{orelse match (r::STAR r::rs) s}
\end{array}
$$

RegExp Matcher

match [] [] = true
match [] _ = f
match (NULL
match (EMPT … rs s
match (CHR c::rs) … tch rs s
match (CHR c::r
match (ALT r … $(r_1$::rs) s
… se match $(r_2$::rs) s
match (SEQ r_1 r_2::rs) s = match $(r_1::r_2$::rs) s
match (STAR r::rs) s = match rs s
orelse match (r::STAR r::rs) s



Second Attempt

Can a regular expression match the empty string?

$$
\begin{aligned}
\text{nullable (NULL)} &= \text{false}\\
\text{nullable (EMPTY)} &= \text{true}\\
\text{nullable (CHR c)} &= \text{false}\\
\text{nullable (ALT } r_1\ r_2) &= (\text{nullable } r_1) \text{ orelse } (\text{nullable } r_2)\\
\text{nullable (SEQ } r_1\ r_2) &= (\text{nullable } r_1) \text{ andalso } (\text{nullable } r_2)\\
\text{nullable (STAR r)} &= \text{true}
\end{aligned}
$$

Second Attempt

Can a regular expression match the empty string?

$$
\begin{array}{lcl}
\text{nullable (NULL)} & = & \text{false} \\
\text{nullable (EMPTY)} & = & \text{true} \\
\text{nullable (CHR c)} & = & \text{false} \\
\text{nullable (ALT } r_1\ r_2\text{)} & = & \text{(nullable } r_1\text{) orelse (nullable } r_2\text{)} \\
\text{nullable (SEQ } r_1\ r_2\text{)} & = & \text{(nullable } r_1\text{) andalso (nullable } r_2\text{)} \\
\text{nullable (STAR r)} & = & \text{true}
\end{array}
$$

✓ ✓ ✓ ✓ ✓ ✓

RegExp Matcher 2

If r matches c::s, which regular expression can match the string s?

$$
\begin{array}{lcl}
\text{der c (NULL)} & = & \text{NULL} \\
\text{der c (EMPTY)} & = & \text{NULL} \\
\text{der c (CHR d)} & = & \text{if c=d then EMPTY else NULL} \\
\text{der c (ALT } r_1\ r_2) & = & \text{ALT (der c } r_1\text{) (der c } r_2) \\
\text{der c (SEQ } r_1\ r_2) & = & \text{ALT (SEQ (der c } r_1\text{) } r_2) \\
 & & \quad \text{(if nullable } r_1 \text{ then der c } r_2 \text{ else NULL)} \\
\text{der c (STAR r)} & = & \text{SEQ (der c r) (STAR r)}
\end{array}
$$

RegExp Matcher 2

If r matches c::s, which regular expression can match the string s?

$$
\begin{array}{lcl}
\text{der c (NULL)} & = & \text{NULL} \\
\text{der c (EMPTY)} & = & \text{NULL} \\
\text{der c (CHR d)} & = & \text{if c=d then EMPTY else NULL} \\
\text{der c (ALT } r_1\ r_2\text{)} & = & \text{ALT (der c } r_1\text{) (der c } r_2\text{)} \\
\text{der c (SEQ } r_1\ r_2\text{)} & = & \text{ALT (SEQ (der c } r_1\text{) } r_2\text{)} \\
 & & \quad \text{(if nullable } r_1 \text{ then der c } r_2 \text{ else NULL)} \\
\text{der c (STAR r)} & = & \text{SEQ (der c r) (STAR r)} \\
\text{derivative r []} & = & \text{r} \\
\text{derivative r (c::s)} & = & \text{derivative (der c r) s}
\end{array}
$$

we call the program with
matches r s = nullable (derivative r s)

RegExp Matcher 2

If r matches c::s, which regular expression can match the string s?

der c (NULL)	=	NULL	✓
der c (EMPTY)	=	NULL	✓
der c (CHR d)	=	if c=d then EMPTY else NULL	✓
der c (ALT r_1 r_2)	=	ALT (der c r_1) (der c r_2)	✓
der c (SEQ r_1 r_2)	=	ALT (SEQ (der c r_1) r_2) (if nullable r_1 then der c r_2 else NULL)	✓
der c (STAR r)	=	SEQ (der c r) (STAR r)	✓
derivative r []	=	r	✓
derivative r (c::s)	=	derivative (der c r) s	✓

we call the program with
matches r s = nullable (derivative r s)

But Now What?

?

Testing

matches $[]^*$ $[]$	$\mapsto$	true
matches $([]\|a)^*$ a	$\mapsto$	true
matches $(a\cdot b)^*$ $[]$	$\mapsto$	true
matches $(a\cdot b)^*$ ab	$\mapsto$	true
matches $(a\cdot b)^*$ aba	$\mapsto$	false
matches $(a\cdot b)^*$ abab	$\mapsto$	true
matches $(a\cdot b)^*$ abaa	$\mapsto$	false
matches $x\cdot(0\|1)^*$ x	$\mapsto$	true
matches $x\cdot(0\|1)^*$ x0	$\mapsto$	true
matches $x\cdot(0\|1)^*$ x3	$\mapsto$	false

Specification

We have to specify what it means for a regular expression to match a string.

Specification

We have to specify what it means for a regular expression to match a string.

$(a \cdot b)^*$

$\quad\mapsto\ \{[], ab, abab, ababab, \ldots\}$

$x \cdot (0 \mid 1 \mid 2 \ldots 8 \mid 9)^*$

$\quad\mapsto\ \{x, x0, x1, \ldots, x00, \ldots, x123, \ldots\}$

Specification

We have to specify what it means for a regular expression to match a string.

$$\begin{aligned}
\mathbb{L}\,(\mathrm{NULL}) &\stackrel{\mathrm{def}}{=} \{\} \\
\mathbb{L}\,(\mathrm{EMPTY}) &\stackrel{\mathrm{def}}{=} \{[]\} \\
\mathbb{L}\,(\mathrm{CHR}\ c) &\stackrel{\mathrm{def}}{=} \{c\} \\
\mathbb{L}\,(\mathrm{ALT}\ r_1\ r_2) &\stackrel{\mathrm{def}}{=} \\
\mathbb{L}\,(\mathrm{SEQ}\ r_1\ r_2) &\stackrel{\mathrm{def}}{=} \\
\mathbb{L}\,(\mathrm{STAR}\ r) &\stackrel{\mathrm{def}}{=}
\end{aligned}$$

Specification

We have to specify what it means for a regular expression to match a string.

$$
\begin{aligned}
\mathbb{L}\,(\mathrm{NULL}) &\stackrel{\text{def}}{=} \{\} \\
\mathbb{L}\,(\mathrm{EMPTY}) &\stackrel{\text{def}}{=} \{[]\} \\
\mathbb{L}\,(\mathrm{CHR}\ c) &\stackrel{\text{def}}{=} \{c\} \\
\mathbb{L}\,(\mathrm{ALT}\ r_1\ r_2) &\stackrel{\text{def}}{=} \mathbb{L}\,(r_1) \cup \mathbb{L}\,(r_2) \\
\mathbb{L}\,(\mathrm{SEQ}\ r_1\ r_2) &\stackrel{\text{def}}{=} \\
\mathbb{L}\,(\mathrm{STAR}\ r) &\stackrel{\text{def}}{=}
\end{aligned}
$$

Specification

We have to specify what it means for a regular expression to match a string.

$$
\begin{aligned}
\mathbb{L}\,(\mathrm{NULL}) &\stackrel{\mathrm{def}}{=} \{\} \\
\mathbb{L}\,(\mathrm{EMPTY}) &\stackrel{\mathrm{def}}{=} \{[]\} \\
\mathbb{L}\,(\mathrm{CHR}\ c) &\stackrel{\mathrm{def}}{=} \{c\} \\
\mathbb{L}\,(\mathrm{ALT}\ r_1\ r_2) &\stackrel{\mathrm{def}}{=} \mathbb{L}\,(r_1) \cup \mathbb{L}\,(r_2) \\
\mathbb{L}\,(\mathrm{SEQ}\ r_1\ r_2) &\stackrel{\mathrm{def}}{=} \\
\mathbb{L}\,(\mathrm{STAR}\ r) &\stackrel{\mathrm{def}}{=}
\end{aligned}
$$

$$S_1\ ;\ S_2 \stackrel{\mathrm{def}}{=} \{\, s_1 @ s_2 \mid s_1 \in S_1 \wedge s_2 \in S_2 \,\}$$

Specification

We have to specify what it means for a regular expression to match a string.

$$
\begin{aligned}
\mathbb{L}\,(\mathrm{NULL}) &\stackrel{\text{def}}{=} \{\} \\
\mathbb{L}\,(\mathrm{EMPTY}) &\stackrel{\text{def}}{=} \{[]\} \\
\mathbb{L}\,(\mathrm{CHR}\ c) &\stackrel{\text{def}}{=} \{c\} \\
\mathbb{L}\,(\mathrm{ALT}\ r_1\ r_2) &\stackrel{\text{def}}{=} \mathbb{L}\,(r_1) \cup \mathbb{L}\,(r_2) \\
\mathbb{L}\,(\mathrm{SEQ}\ r_1\ r_2) &\stackrel{\text{def}}{=} \mathbb{L}\,(r_1)\ ;\ \mathbb{L}\,(r_2) \\
\mathbb{L}\,(\mathrm{STAR}\ r) &\stackrel{\text{def}}{=}
\end{aligned}
$$

$$S_1\ ;\ S_2 \stackrel{\text{def}}{=} \{\, s_1 @ s_2 \mid s_1 \in S_1 \wedge s_2 \in S_2 \,\}$$

Specification

We have to specify what it means for a regular expression to match a string.

$$
\begin{aligned}
\mathbb{L}\,(\mathrm{NULL}) &\stackrel{\text{def}}{=} \{\} \\
\mathbb{L}\,(\mathrm{EMPTY}) &\stackrel{\text{def}}{=} \{[]\} \\
\mathbb{L}\,(\mathrm{CHR}\ c) &\stackrel{\text{def}}{=} \{c\} \\
\mathbb{L}\,(\mathrm{ALT}\ r_1\ r_2) &\stackrel{\text{def}}{=} \mathbb{L}\,(r_1) \cup \mathbb{L}\,(r_2) \\
\mathbb{L}\,(\mathrm{SEQ}\ r_1\ r_2) &\stackrel{\text{def}}{=} \mathbb{L}\,(r_1) \,;\, \mathbb{L}\,(r_2) \\
\mathbb{L}\,(\mathrm{STAR}\ r) &\stackrel{\text{def}}{=}
\end{aligned}
$$

$$
\frac{}{[] \in S^\star} \qquad \frac{s_1 \in S \quad s_2 \in S^\star}{s_1 @ s_2 \in S^\star}
$$

Specification

We have to specify what it means for a regular expression to match a string.

$$\begin{aligned}
\mathbb{L}\,(\mathrm{NULL}) &\stackrel{\text{def}}{=} \{\} \\
\mathbb{L}\,(\mathrm{EMPTY}) &\stackrel{\text{def}}{=} \{[]\} \\
\mathbb{L}\,(\mathrm{CHR}\ c) &\stackrel{\text{def}}{=} \{c\} \\
\mathbb{L}\,(\mathrm{ALT}\ r_1\ r_2) &\stackrel{\text{def}}{=} \mathbb{L}\,(r_1) \cup \mathbb{L}\,(r_2) \\
\mathbb{L}\,(\mathrm{SEQ}\ r_1\ r_2) &\stackrel{\text{def}}{=} \mathbb{L}\,(r_1) \,;\, \mathbb{L}\,(r_2) \\
\mathbb{L}\,(\mathrm{STAR}\ r) &\stackrel{\text{def}}{=} (\mathbb{L}\,(r))^\star
\end{aligned}$$

$$\frac{}{[] \in S^\star} \qquad \frac{s_1 \in S \quad s_2 \in S^\star}{s_1 @ s_2 \in S^\star}$$

Is the Matcher Error-Free?

We expect that

$$\text{matches } r\ s = \text{true} \quad \Longrightarrow \quad s \in \mathbb{L}(r)$$
$$\text{matches } r\ s = \text{false} \quad \Longrightarrow \quad s \notin \mathbb{L}(r)$$

Is the Matcher Error-Free?

We expect that

$$\text{matches r s = true} \quad \Longleftarrow \quad s \in \mathbb{L}(r)$$
$$\text{matches r s = false} \quad \Longleftarrow \quad s \notin \mathbb{L}(r)$$

Is the Matcher Error-Free?

We expect that

$$\text{matches r s = true} \iff s \in \mathbb{L}(r)$$
$$\text{matches r s = false} \iff s \notin \mathbb{L}(r)$$

By induction, we can **prove** these properties.

Lemmas:

$$\text{nullable}(r) \iff [] \in \mathbb{L}(r)$$
$$s \in \mathbb{L}(\text{der c r}) \iff (c::s) \in \mathbb{L}(r)$$

Is the Matcher Error-Free?

We expect that

$\forall$ r s.

$$\text{matches r s = true} \iff s \in \mathbb{L}(r)$$

$$\text{matches r s = false} \iff s \notin \mathbb{L}(r)$$

By induction, we can **prove** these properties.

Lemmas:

$$\text{nullable}(r) \iff [] \in \mathbb{L}(r)$$

$$s \in \mathbb{L}(\text{der c r}) \iff (c::s) \in \mathbb{L}(r)$$

$$
\begin{array}{lcl}
\text{nullable (NULL)} & = & \text{false} \\
\text{nullable (EMPTY)} & = & \text{true} \\
\text{nullable (CHR c)} & = & \text{false} \\
\text{nullable (ALT } r_1\ r_2) & = & (\text{nullable } r_1) \text{ orelse } (\text{nullable } r_2) \\
\text{nullable (SEQ } r_1\ r_2) & = & (\text{nullable } r_1) \text{ andalso } (\text{nullable } r_2) \\
\text{nullable (STAR r)} & = & \text{true}
\end{array}
$$

$$
\begin{array}{lcl}
\text{der c (NULL)} & = & \text{NULL} \\
\text{der c (EMPTY)} & = & \text{NULL} \\
\text{der c (CHR d)} & = & \text{if c=d then EMPTY else NULL} \\
\text{der c (ALT } r_1\ r_2) & = & \text{ALT (der c } r_1) \text{ (der c } r_2) \\
\text{der c (SEQ } r_1\ r_2) & = & \text{ALT (SEQ (der c } r_1) \ r_2) \\
 & & \quad (\text{if nullable } r_1 \text{ then der c } r_2 \text{ else NULL}) \\
\text{der c (STAR r)} & = & \text{SEQ (der c r) (STAR r)}
\end{array}
$$

$$
\begin{array}{lcl}
\text{derivative r []} & = & \text{r} \\
\text{derivative r (c::s)} & = & \text{derivative (der c r) s}
\end{array}
$$

$$
\text{matches r s} = \text{nullable (derivative r s)}
$$

```
nullable (NULL)       =  false
nullable (EMPTY)      =  true
nullable (CHR c)      =  false
nullable (ALT r1 r2)  =  (nullable r1) orelse (nullable r2)
nullable (SEQ r1 r2)  =  (nullable r1) andalso (nullable r2)
nullable (STAR r)     =  true

der c (NULL)        =  NULL
der c (EMPTY)       =  NULL
der c (CHR d)       =  if c=d then EMPTY else NULL
der c (ALT r1 r2)   =  ALT (der c r1) (der c r2)
der c (SEQ r1 r2)   =  ALT (SEQ (der c r1) r2)
                           (if nullable r1 then der c r2 else NULL)
der c (STAR r)      =  SEQ (der c r) (STAR r)

derivative r []      =  r
derivative r (c::s)  =  derivative (der c r) s

matches r s = nullable (derivative r s)
```

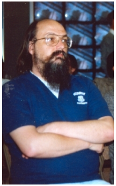
Interlude: TCB

- The **Trusted Code Base** (TCB) is the code that can make your program behave in unintended ways (i.e. cause bugs).
- Typically the TCB includes: CPUs, operating systems, C-libraries, device drivers, (J)VMs...

Interlude: TCB

- The **Trusted Code Base** (TCB) is the code that can make your program behave in unintended ways (i.e. cause bugs).
- Typically the TCB includes: CPUs, operating systems, C-libraries, device drivers, (J)VMs...
- It also includes the compiler.

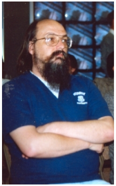
Hacking Compilers



Ken Thompson
Turing Award, 1983

- Ken Thompson showed how to hide a Trojan Horse in a compiler without leaving any traces in the source code.
- No amount of source level verification will protect you from such Thompson-hacks.
- Therefore in safety-critical systems it is important to rely on only a very small TCB.

Hacking Compilers

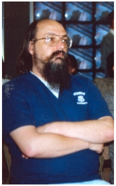


Ken Thompson
Turing Award, 198

1) *Assume you ship the compiler as binary and also with sources.*
2) *Make the compiler aware when it compiles itself.*
3) *Add the Trojan horse.*
4) *Compile.*
5) *Delete Trojan horse from the sources of the compiler.*
6) *Go on holiday for the rest of your life. ;o)*

le
ut
e.
fi-
h
it
ry

Hacking Compilers



Ken Thompson
Turing Award, 1983

- Ken Thompson showed how to hide a Trojan Horse in a compiler without leaving any traces in the source code.
- No amount of source level verification will protect you from such Thompson-hacks.
- Therefore in safety-critical systems it is important to rely on only a very small TCB.

An Example: Small TCB for A Critical Infrastructure

Andrew Appel
(Princeton)

Proof-Carrying Code

code developer/ web server/ Apple Store

code →

user needs to run untrusted code

An Example: Small TCB for A Critical Infrastructure

Andrew Appel (Princeton)

Proof-Carrying Code

code developer/ web server/ Apple Store

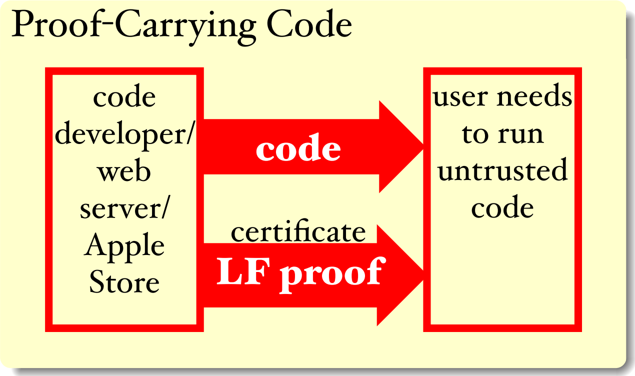
code

user needs to run untrusted code

Highly Dangerous!

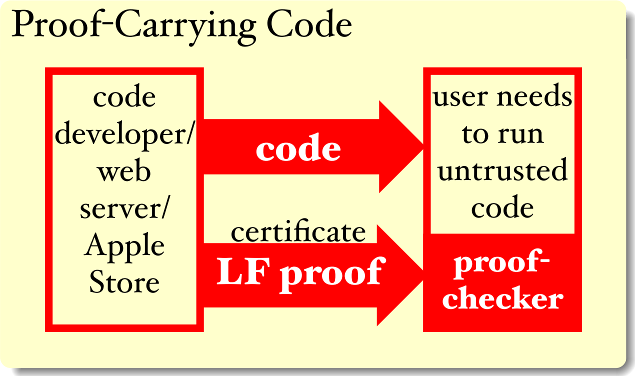
An Example: Small TCB for A Critical Infrastructure

Andrew Appel
(Princeton)



An Example: Small TCB for A Critical Infrastructure

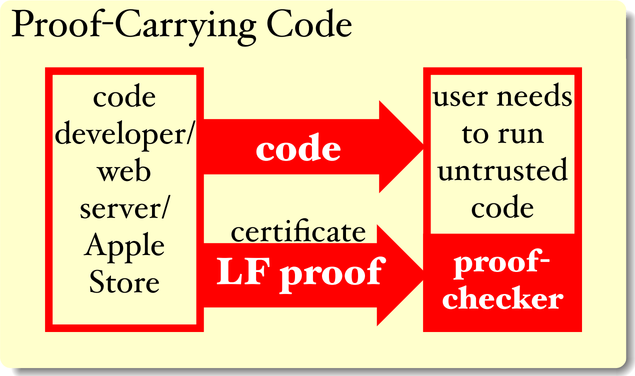
Andrew Appel
(Princeton)



- TCB of the checker is ∼2700 lines of code (1865 loc of LF definitions; 803 loc in C including 2 library functions)

An Example: Small TCB for A Critical Infrastructure

Andrew Appel (Princeton)



- TCB of the 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

Type-Checking in LF

Spec ← Proof → Alg

Bob Harper
(CMU)

Frank Pfenning
(CMU)

31 pages in
ACM Transact. on
Comp. Logic., 2005

Type-Checking in LF

Spec ← Proof → Alg

Bob Harper
(CMU)

Frank Pfenning
(CMU)

31 pages in
ACM Transact. on
Comp. Logic., 2005

Type-Checking in LF

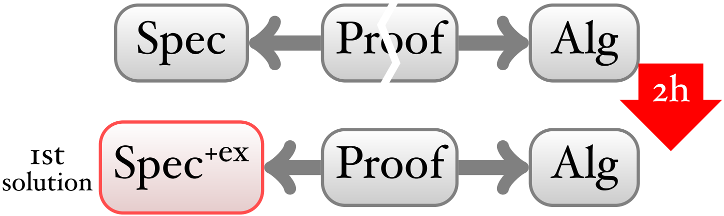
Spec ← Proof → Alg

Bob Harper
(CMU)

Frank Pfenning
(CMU)

31 pages in
ACM Transact. on
Comp. Logic., 2005

Type-Checking in LF



Bob Harper
(CMU)

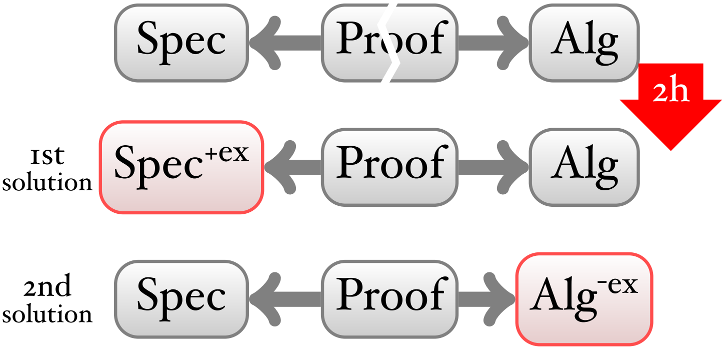
Frank Pfenning
(CMU)

31 pages in
ACM Transact. on
Comp. Logic., 2005

Type-Checking in LF

Bob Harper
(CMU)

Frank Pfenning
(CMU)

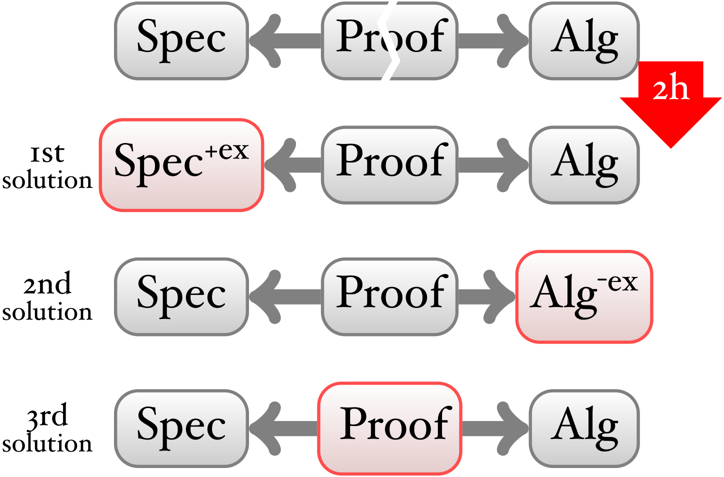


31 pages in
ACM Transact. on
Comp. Logic., 2005

Type-Checking in LF

Bob Harper
(CMU)

Frank Pfenning
(CMU)



31 pages in
ACM Transact. on
Comp. Logic., 2005

Type-Checking in LF

Spec ← Proof → Alg

2h

Bob Harper
(CMU)

1st solution: Spec^{+ex} ← Proof → Alg

2nd solution: Spec ← Proof → Alg^{-ex}

Frank Pfenning
(CMU)

3rd solution: Spec ← Proof → Alg

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

Theorem Provers

- Theorem provers help with keeping large proofs consistent; make them modifiable.
- They can ensure that all cases are covered.
- Sometimes, tedious reasoning can be automated.

Theorem Provers

- You also pay a (sometimes heavy) price: reasoning can be much, much harder.
- Depending on your domain, suitable reasoning infrastructure might not yet be available.

Theorem Provers

Recently impressive work has been accomplished proving the correctness

- of a compiler for C-light (compiled code has the same observable behaviour as the original source code),
- a mirco-kernel operating system (absence of certain bugs...no nil pointers, no buffer overflows).

Trust in Theorem Provers

Why should we trust theorem provers?

Theorem Provers

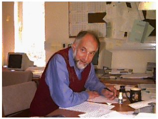
- Theorem provers are a special kind of software.
- We do **not** need to trust them; we only need to trust:
  - *The logic they are based on (e.g. HOL), and*
  - *a proof checker that checks the proofs (this can be a very small program).*

Theorem Provers

- Theorem provers are a special kind of software.
- We do **not** need to trust them; we only need to trust:
  - *The logic they are based on (e.g. HOL), and*
  - *a proof checker that checks the proofs (this can be a very small program).*
  - *To a little extend, we also need to trust our definitions (this can be mitigated).*

Isabelle

- I am using the Isabelle theorem prover (development since 1990).



Robin Milner
Turing Award, 1991

- It follows the LCF-approach:
  - Have a special abstract type **thm**.
  - Make the constructors of this abstract type the inference rules of your logic.
  - Implement the theorem prover in a strongly-typed language (e.g. ML).

$\Rightarrow$ everything of type **thm** has been proved (even if we do not have to explicitly generate proofs).

Demo

Future Research

- Make theorem provers more like a programming environment.

Future Research

- Make theorem provers more like a programming environment.
- Use all the computational power we get from the hardware to automate reasoning (GPUs).

Future Research

- Make theorem provers more like a programming environment.
- Use all the computational power we get from the hardware to automate reasoning (GPUs).
- Provide a comprehensive reasoning infrastructure for many domains and design automated decision procedures.

Future Research

- Make theorem provers more like a programming environment.
- Use all the computational power we get from the hardware to automate reasoning (GPUs).
- Provide a comprehensive reasoning infrastructure for many domains and design automated decision procedures.

> "Formal methods will never have a significant impact until they can be used by people that don't understand them."
>
> attributed to Tom Melham

Conclusion

- The plan is to make this kind of programming the "future".

Conclusion

- The plan is to make this kind of programming the "future".
- Though the technology is already there (compiler + micro-kernel os).

Conclusion

- The plan is to make this kind of programming the "future".
- Though the technology is already there (compiler + micro-kernel os).
- Logic and reasoning (especially induction) are important skills for Computer Scientists.

Thank you very much!

Questions?