Leapfrog:
Certified Equivalence for Protocol Parsers

Tobias Kappé

Mathematical & Computational Logic

ILLC Midwinter Colloquium
December 14th, 2021

Joint work with folks at Cornell

Ryan Doenges

John Sarracino

Nate Foster

Greg Morrisett

Packet parsing

```
01000111011011110010
00000110001001101001
01100111001000000111
00100110010101100100
```

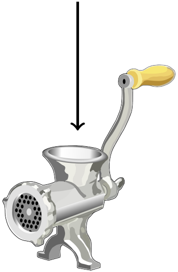
Packet parsing

01000111011011110010

00000110001001101001

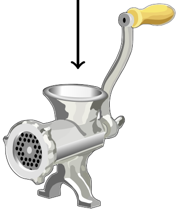
01100111001000000111

00100110010101100100



Packet parsing

01000111011011110010
00000110001001101001
01100111001000000111
00100110010101100100



Success

```
header baby_ip {
 bit<8> src;
 bit<8> dst;
 bit<4> proto;
}
```

Packet parsing

01000111011011110010

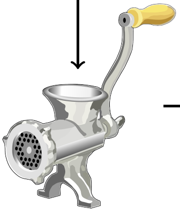
00000110001001101001

01100111001000000111

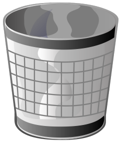
00100110010101100100

```
header baby_ip {
 bit<8> src;
 bit<8> dst;
 bit<4> proto;
}
```

Success



Error



Updating the parser

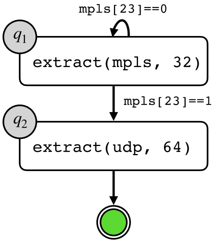
Updating the parser

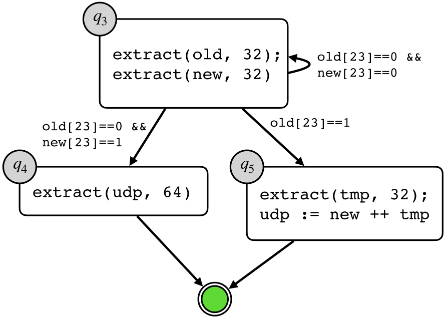
 

Updating the parser

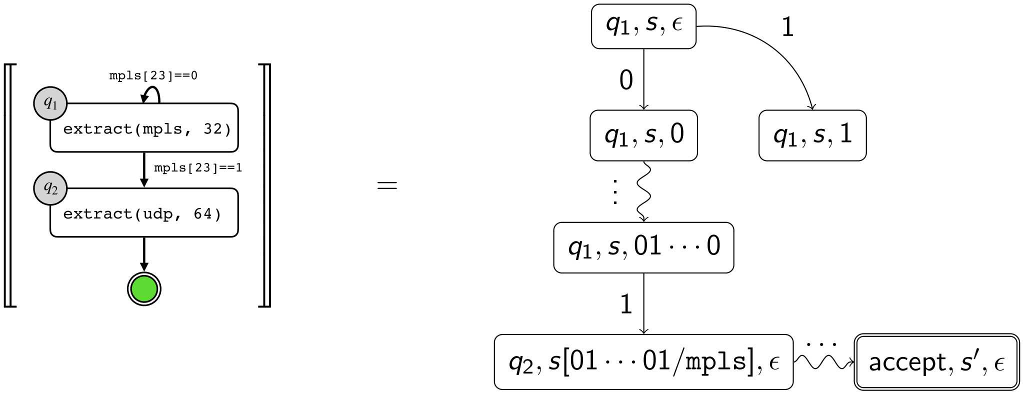
$$\stackrel{?}{\approx}$$

Running Example



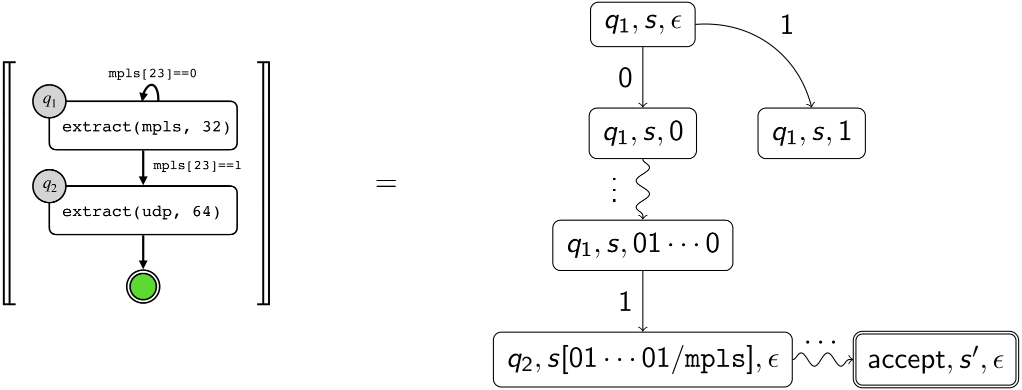


Semantics



mpls[23]==0
q_1
extract(mpls, 32)
mpls[23]==1
q_2
extract(udp, 64)
=
q_1, s, ϵ
1
0
$q_1, s, 0$
$q_1, s, 1$
$q_1, s, 01 \cdots 0$
1
$q_2, s[01 \cdots 01/\texttt{mpls}], \epsilon$
$accept, s', \epsilon$

Semantics



$$L(c) = \{w \in \{0,1\}^* : c \xrightarrow{w} \langle \text{accept}, s', \epsilon \rangle\}$$

Challenge

Definition (Bisimulation)

A *bisimulation* is a relation R on configurations such that for all $c_1 \; R \; c_2$:

1. c_1 is accepting if and only if c_2 is accepting
2. if $c_1 \xrightarrow{b} c_1'$ and $c_2 \xrightarrow{b} c_2'$, then $c_1' \; R \; c_2'$.

Challenge

Definition (Bisimulation)

A *bisimulation* is a relation R on configurations such that for all $c_1 \ R \ c_2$:

1. c_1 is accepting if and only if c_2 is accepting
2. if $c_1 \xrightarrow{b} c_1'$ and $c_2 \xrightarrow{b} c_2'$, then $c_1' \ R \ c_2'$.

Lemma (Bisimilarity characterizes language equivalence)

$L(c_1) = L(c_2)$ *if and only if* $c_1 \ R \ c_2$ *for some bisimulation* R*.*

Challenge

Definition (Bisimulation)

A *bisimulation* is a relation R on configurations such that for all $c_1 \; R \; c_2$:

1. c_1 is accepting if and only if c_2 is accepting
2. if $c_1 \xrightarrow{b} c_1'$ and $c_2 \xrightarrow{b} c_2'$, then $c_1' \; R \; c_2'$.

Lemma (Bisimilarity characterizes language equivalence)

$L(c_1) = L(c_2)$ *if and only if* $c_1 \; R \; c_2$ *for some bisimulation* R*.*

Problem: $|\text{configurations}| \geq 10^{37}$ for reference MPLS parser.

- Symbolic representation + SMT solving.
- *"Up-to" techniques* to skip buffering.

Symbolic representation

First-order logic with semantics $\llbracket \phi \rrbracket \subseteq$ configurations $\times$ configurations.

Examples

- $\phi = q_1^<$ means "the left state is q_1"

Symbolic representation

First-order logic with semantics $[\![\phi]\!] \subseteq \text{configurations} \times \text{configurations}$.

Examples

- $\phi = q_1^<$ means "the left state is q_1"
- $\texttt{mpls}^<[24:24] = 1$ means "the 24th bit of `mpls` (on the left) is 1"

Symbolic representation

First-order logic with semantics $[\![\phi]\!] \subseteq$ configurations $\times$ configurations.

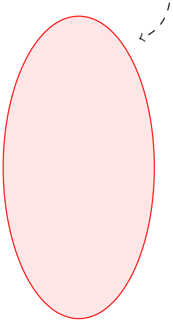
Examples

- $\phi = q_1^<$ means "the left state is q_1"
- $\texttt{mpls}^<[24:24] = 1$ means "the 24th bit of `mpls` (on the left) is 1"

If $[\![\phi]\!]$ is a bisimulation, then ϕ is a *symbolic bisimulation*.

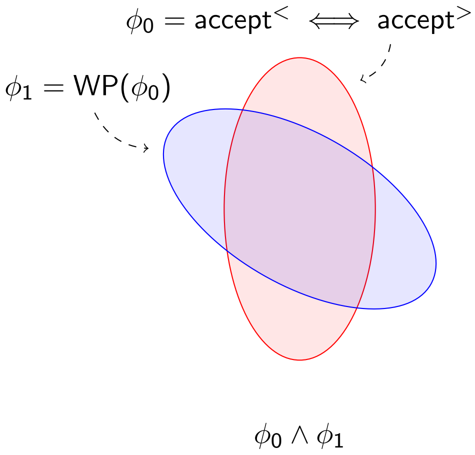
Equivalence checking — intuition

$$\phi_0 = \text{accept}^< \iff \text{accept}^>$$

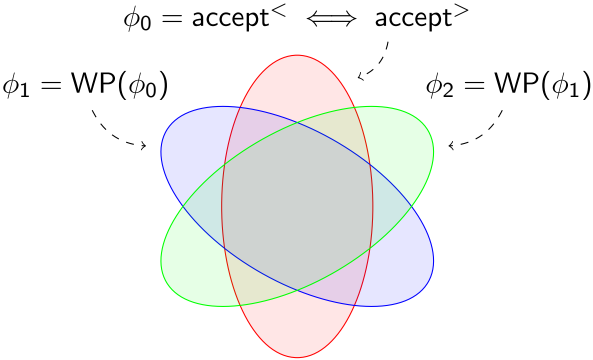


ϕ_0

Equivalence checking — intuition



Equivalence checking — intuition



$\phi_0 \wedge \phi_1 \wedge \phi_2$

Equivalence checking — algorithm

$R \leftarrow \emptyset$
$T \leftarrow \{\text{accept}^{<} \iff \text{accept}^{>}\}$
while $T \neq \emptyset$ **do**
  pop ψ from T
  if not $\bigwedge R \vDash \psi$ **then**
    $R \leftarrow R \cup \{\psi\}$
    $T \leftarrow T \cup \text{WP}(\psi)$
if $\phi \vDash \bigwedge R$ **then**
  return true
else
  return false

Equivalence checking — algorithm

$R \leftarrow \emptyset$
$T \leftarrow \{\text{accept}^{<} \iff \text{accept}^{>}\}$
while $T \neq \emptyset$ **do**
  pop ψ from T
  if not $\bigwedge R \vDash \psi$ **then**
    $R \leftarrow R \cup \{\psi\}$
    $T \leftarrow T \cup \text{WP}(\psi)$
if $\phi \vDash \bigwedge R$ **then**
  return true
else
  return false

Loop termination: either

- $[\![\bigwedge R]\!]$ shrinks; or
- $[\![\bigwedge R]\!]$ stays the same, T shrinks.

Equivalence checking — algorithm

$R \leftarrow \emptyset$
$T \leftarrow \{\mathsf{accept}^{<} \iff \mathsf{accept}^{>}\}$
while $T \neq \emptyset$ **do**
    pop ψ from T
    if not $\bigwedge R \vDash \psi$ **then**
        $R \leftarrow R \cup \{\psi\}$
        $T \leftarrow T \cup \mathsf{WP}(\psi)$
if $\phi \vDash \bigwedge R$ **then**
    return true
else
    return false

Loop termination: either

- $[\![\bigwedge R]\!]$ shrinks; or
- $[\![\bigwedge R]\!]$ stays the same, T shrinks.

After the loop, $\bigwedge R$ is the *weakest* symbolic bisimulation.

Implementation

Implementation — Side-stepping the termination checker

Implementation — Side-stepping the termination checker

Algorithm state as proof rules:

$$\frac{\phi \models \bigwedge R}{\texttt{pre\_bisim}\ \phi\ R\ \texttt{[]}}\ \textsc{close} \qquad \frac{\bigwedge R \models \psi \qquad \texttt{pre\_bisim}\ \phi\ R\ T}{\texttt{pre\_bisim}\ \phi\ R\ (\psi :: T)}\ \textsc{Skip}$$

$$\frac{\bigwedge R \not\models \psi \qquad \texttt{pre\_bisim}\ \phi\ (\psi :: R)\ (T; \mathsf{WP}(\psi))}{\texttt{pre\_bisim}\ \phi\ R\ (\psi :: T)}\ \textsc{Extend}$$

Implementation — Side-stepping the termination checker

Algorithm state as proof rules:

$$\frac{\phi \vDash \bigwedge R}{\texttt{pre\_bisim}\ \phi\ R\ \texttt{[]}}\ \textsc{close} \qquad \frac{\bigwedge R \vDash \psi \qquad \texttt{pre\_bisim}\ \phi\ R\ T}{\texttt{pre\_bisim}\ \phi\ R\ (\psi :: T)}\ \textsc{Skip}$$

$$\frac{\bigwedge R \nvDash \psi \qquad \texttt{pre\_bisim}\ \phi\ (\psi :: R)\ (T; \mathsf{WP}(\psi))}{\texttt{pre\_bisim}\ \phi\ R\ (\psi :: T)}\ \textsc{Extend}$$

Lemma (Soundness)

If $\texttt{pre\_bisim}\ \phi\ \texttt{[]}\ I$*, then all pairs in* $[\![\phi]\!]$ *are bisimilar.*

Implementation — Side-stepping the termination checker

Algorithm state as proof rules:

$$\frac{\phi \models \bigwedge R}{\texttt{pre\_bisim}\ \phi\ R\ []}\ \textsc{close} \qquad \frac{\bigwedge R \models \psi \qquad \texttt{pre\_bisim}\ \phi\ R\ T}{\texttt{pre\_bisim}\ \phi\ R\ (\psi :: T)}\ \textsc{Skip}$$

$$\frac{\bigwedge R \not\models \psi \qquad \texttt{pre\_bisim}\ \phi\ (\psi :: R)\ (T; \mathsf{WP}(\psi))}{\texttt{pre\_bisim}\ \phi\ R\ (\psi :: T)}\ \textsc{Extend}$$

Lemma (Soundness)

If $\texttt{pre\_bisim}\ \phi\ []\ I$, *then all pairs in* $[\![\phi]\!]$ *are bisimilar.*

Workflow: proof search for `pre_bisim`, applying exactly one of these three rules.

Implementation — Talk to SMT solver

Implementation — Talk to SMT solver

In theory:

- If T is empty, apply `Done`.
- If $\bigwedge R \vDash \psi$, apply `Skip`.
- If $\bigwedge R \nvDash \psi$, apply `Extend`.

In practice:

- Massage entailment into fully quantified boolean formula.
- Custom plugin pretty-prints to SMT-LIB 2.0, asks solver.
- If `SAT`, admit $\bigwedge R \vDash \psi$ and apply `Skip`.
- If `UNSAT`, admit $\bigwedge R \nvDash \psi$ and apply `Extend`.

Evaluation — Microbenchmarks

Automatically verifies common transformations:

- Speculative extraction / vectorization.
- Common prefix factorization
- General versus specialized TLV parsing.
- Early versus late filtering.

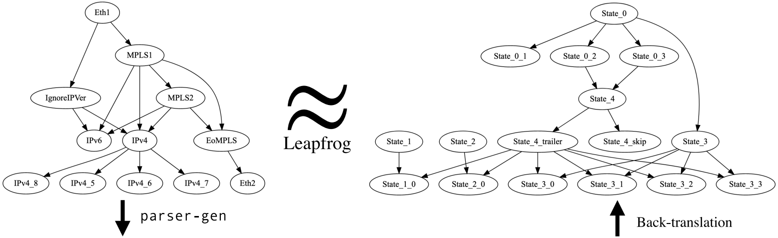
Extends to certain hyperproperties:

- Independence of initial header store.
- Correspondence between final stores.

Evaluation — Applicability study

`parser-gen` (Gibb et al. 2013) compiles parser to optimized implementation.

- Benchmarks: about 30 states each, *huge* store datastructure.
- Leapfrog can validate equivalence of input to output.




```
Match: ([ff, 00, 00, ff, ff, 00, 00, 00, 00], [00, 00, 00, 08, 00, 00, 00, 00, 00]) Next-State: 3/255   Adv:  14  Next-Lookup: [0, 0, 0, 0]
Match: ([ff, 00, 00, ff, ff, 00, 00, 00, 00], [00, 00, 00, 88, 47, 00, 00, 00, 00]) Next-State: 4/255   Adv:  16  Next-Lookup: [0, 2, 4, 6]
Match: ([ff, 01, 00, 00, 00, 01, 00, f0, 00], [04, 00, 00, 00, 00, 01, 00, 00, 00]) Next-State: 1/255   Adv:   6  Next-Lookup: [0, 0, 0, 0]
Match: ([ff, 01, 00, f0, 00, 00, 00, 00, 00], [04, 01, 00, 00, 00, 00, 00, 00, 00]) Next-State: 1/255   Adv:   2  Next-Lookup: [0, 0, 0, 0]
...
Match: ([ff, 00, 00, 00, 00, 00, 00, 00, 00], [04, 00, 00, 00, 00, 00, 00, 00, 00]) Next-State: 255/255 Adv:  2   Next-Lookup: [0, 0, 0, 0]
```

Lessons learned

- Finite automata can go the distance.
- SMT solvers are *really* powerful.
- Programming in Coq is fun.

http://langsec.org/occupy/

References

G. Gibb et al. (2013). "Design principles for packet parsers". In: *ANCS*, pp. 13–24. DOI: 10.1109/ANCS.2013.6665172.