Here be Wyverns!
Verifying LLVM-IR with llStar

Jules Villard

Imperial College London

http://llstar.pauvre.org/
- LLVM is a compiler framework (front-ends, optimisations, JIT, assembler, ...)
- Common intermediate language: LLVM-IR
- Front-ends to C, C++, Fortran, ...
Overview of llStar

- LLVM-IR
- Specs
- User input
- yes/fail
- LLVM rules
- user rules
- coreStar
- SMT solver

User input → input → output → tool interaction
The LLVM module to analyse (LLVM 3.4)
The LLVM module to analyse (LLVM 3.4)

Separation logic pre/post-conditions for functions
The LLVM module to analyse (LLVM 3.4)

Separation logic pre/post-conditions for functions

Symbolic execution and frame inference engine for separation logic.
The LLVM module to analyse (LLVM 3.4)

Separation logic pre/post-conditions for functions

Symbolic execution and frame inference engine for separation logic.

Reasoning rules for pointers, structures, etc.
The LLVM module to analyse (LLVM 3.4)

Separation logic pre/post-conditions for functions

Symbolic execution and frame inference engine for separation logic.

Reasoning rules for pointers, structures, etc.

Reasoning rules for inductive data structures.
The LLVM module to analyse (LLVM 3.4)

Separation logic pre/post-conditions for functions

Symbolic execution and frame inference engine for separation logic.

Reasoning rules for pointers, structures, etc.

Reasoning rules for inductive data structures.

Z3 checks formulas over pure values (bit-vectors, structs, arrays, floats)
The LLVM module to analyse (LLVM 3.4)

Separation logic pre/post-conditions for functions

Symbolic execution and frame inference engine for separation logic.

Reasoning rules for pointers, structures, etc.

Reasoning rules for inductive data structures.

Z3 checks formulas over pure values (bit-vectors, structs, arrays, floats)

proved specs + memory safety / failed proof
Symbolic states

llStar and coreStar use separation logic formulas to describe the state of the heap and variables during program execution.

\[ A ::= e_1 = e_2 \mid e_1 \neq e_2 \mid e_1 >_{sbv} e_2 \mid \ldots \quad \text{pure predicates} \]
\[ \mid \text{emp} \quad \text{empty heap} \]
\[ \mid e_p \xrightarrow{t} e_v \quad \text{points-to (with type)} \]
\[ \mid \text{malloced}(e_p, e_s) \quad \text{malloc block (with size)} \]
\[ \mid \ldots \quad \text{more spatial predicates} \]
\[ \mid A_1 \ast A_2 \mid A_1 \lor A_2 \quad \text{logical connectives} \]
Symbolic states

IlStar and coreStar use separation logic formulas to describe the state of the heap and variables during program execution.

\[ A ::= e_1 = e_2 \mid e_1 \neq e_2 \mid e_1 >_{sbv} e_2 \mid \ldots \quad \text{pure predicates} \\
| \text{emp} \quad \text{empty heap} \\
| e_p \xrightarrow{t} e_v \quad \text{points-to (with type)} \\
| \text{malloced} (e_p, e_s) \quad \text{malloc block (with size)} \\
| \ldots \\
| A_1 \ast A_2 \mid A_1 \lor A_2 \quad \text{more spatial predicates} \\
| A_1 \ast A_2 \mid A_1 \lor A_2 \quad \text{logical connectives} \\

- Pure predicates: values only, no heap
Symbolic states

llStar and coreStar use separation logic formulas to describe the state of the heap and variables during program execution.

\[
A ::= e_1 = e_2 \mid e_1 \neq e_2 \mid e_1 >_{sbv} e_2 \mid \ldots \quad \text{pure predicates}
\]
\[
\mid \text{emp}
\]
\[
\mid e_p \leftarrow^{t} e_v
\]
\[
\mid \text{malloced}(e_p, e_s)
\]
\[
\mid \ldots
\]
\[
\mid A_1 \ast A_2 \mid A_1 \lor A_2
\]

- Pure predicates: values only, no heap
- Classical conjunction $\land$ and SL conjunction $\ast$ coincide on pure facts:

\[
x = \text{bv\_const}(32, 4) \ast y = x \iff x = \text{bv\_const}(32, 4) \land y = x
\]
• Splinter tool from jStar [Distefano, Parkinson’08] (SL for Java)

LLVM integers and operations (add/mul/shl/. . . ) \Rightarrow \text{bitvectors and bitvector arithmetic}

LLVM structs \Rightarrow \text{records}

LLVM arrays \Rightarrow \text{arrays}
• Splinter tool from jStar [Distefano, Parkinson’08] (SL for Java)
• Focuses on Java-independent reasoning
- Splinter tool from jStar [Distefano, Parkinson’08] (SL for Java)
- Focuses on Java-independent reasoning
- Originally developed by Botincan, Distefano, Dodds, Grigore, Naudziuniene, Parkinson

New!
Extended to reason about LLVM values as translated by llStar:
- LLVM integers and operations (add/mul/shl/\ldots) \Rightarrow bitvectors and bitvector arithmetic
- LLVM structs \Rightarrow records
- LLVM arrays \Rightarrow arrays

Reasoning about values delegated to Z3
• Splinter tool from jStar [Distefano, Parkinson’08] (SL for Java)
• Focuses on Java-independent reasoning
• Originally developed by Botincan, Distefano, Dodds, Grigore, Naudziuniene, Parkinson
• **New!** Extended to reason about LLVM values as translated by llStar:

  - LLVM integers and operations (add/mul/shl/...) \(\Rightarrow\) bitvectors and bitvector arithmetic
  - LLVM structs \(\Rightarrow\) records
  - LLVM arrays \(\Rightarrow\) arrays

Reasoning about values delegated to Z3
• Splinter tool from jStar [Distefano, Parkinson’08] (SL for Java)
• Focuses on Java-independent reasoning
• Originally developed by Botincan, Distefano, Dodds, Grigore, Naudziuniene, Parkinson
• **New!** Extended to reason about LLVM values as translated by llStar:
  ○ LLVM integers and operations (add/mul/shl/…)
    ⇝ bitvectors and bitvector arithmetic
• Splinter tool from jStar [Distefano, Parkinson’08] (SL for Java)
• Focuses on Java-independent reasoning
• Originally developed by Botincan, Distefano, Dodds, Grigore, Naudziuniene, Parkinson
• **New!** Extended to reason about LLVM values as translated by llStar:
  o LLVM integers and operations ($\text{add/mul/shl}/...$) $\mapsto$ bitvectors and bitvector arithmetic
  o LLVM structs $\mapsto$ records
• Splinter tool from jStar [Distefano, Parkinson’08] (SL for Java)
• Focuses on Java-independent reasoning
• Originally developed by Botincan, Distefano, Dodds, Grigore, Naudziuniene, Parkinson
• **New!** Extended to reason about LLVM values as translated by llStar:
  - LLVM integers and operations (\texttt{add/mul/shl/...}) \rightarrow bitvectors and bitvector arithmetic
  - LLVM structs \rightarrow records
  - LLVM arrays \rightarrow arrays
• Splinter tool from jStar [Distefano, Parkinson’08] (SL for Java)
• Focuses on Java-independent reasoning
• Originally developed by Botincan, Distefano, Dodds, Grigore, Naudziuniene, Parkinson
• **New!** Extended to reason about LLVM values as translated by llStar:
  - LLVM integers and operations ($\text{add/mul/shl}/...$) $\mapsto$ bitvectors and bitvector arithmetic
  - LLVM structs $\mapsto$ records
  - LLVM arrays $\mapsto$ arrays
• Reasoning about values delegated to Z3
# From LLVM-IR to coreStar

## coreStar input language

| CI | ::= | label(l) | block label |
| --- | | goto(l_1, ..., l_n) | non-deterministic jump |
| x = {A}{B}(e_0, ..., e_N) | Hoare triple |

(A, B separation logic formulas with free variables @parameter0: ..., @parameterN: and $ret)
coreStar input language

\[ CI ::= \text{label}(l) \quad \text{block label} \]
\[ | \text{goto}(l_1, \ldots, l_n) \quad \text{non-deterministic jump} \]
\[ | x = \{A\} \{B\}(e_0, \ldots, e_N) \quad \text{Hoare triple} \]

(A, B separation logic formulas with free variables @parameter0:, ..., @parameterN: and $ret)

\text{llStar translates an LLVM module to a coreStar CFG:}
**coreStar input language**

<table>
<thead>
<tr>
<th>CI</th>
<th>::= label(I)</th>
<th>block label</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>goto(I₁,…,Iₙ)</td>
<td>non-deterministic jump</td>
</tr>
<tr>
<td></td>
<td>x = {A}{B}(e₀,…,eₙ)</td>
<td>Hoare triple</td>
</tr>
</tbody>
</table>

\( (A, B \text{ separation logic formulas with free variables} \)

\( @\text{parameter}_0:,…, @\text{parameter}_N: \text{ and } \$\text{ret} ) \)

llStar translates an LLVM module to a coreStar CFG:

\[
x = \text{add } t \ v₁, \ v₂ \rightsquigarrow x = \{\text{emp}\}\{\$\text{ret} = \text{bvadd}(v₁, v₂)\}
\]
From LLVM-IR to coreStar

**coreStar input language**

<table>
<thead>
<tr>
<th>CI</th>
<th>:= label(l)</th>
<th>block label</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>goto(l₁,...,lₙ)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x = {A}{B}(e₀,...,eₙ)</td>
</tr>
</tbody>
</table>

(A, B separation logic formulas with free variables
@parameter₀:,...,@parameterₜ: and $ret)

llStar translates an LLVM module to a coreStar CFG:

```
x = add t v₁, v₂ ⇝ x = {emp}{$ret = bvadd(v₁, v₂)}
unreachable ⇝ {false}{false}
```
From LLVM-IR to coreStar

## coreStar input language

\[
CI ::= \text{label}(l) \quad \text{block label}
| \text{goto}(l_1, \ldots, l_n) \quad \text{non-deterministic jump}
| x = \{A\}\{B\}(e_0, \ldots, e_N) \quad \text{Hoare triple}
\]

\(A, B\) separation logic formulas with free variables

@parameter0: \ldots, @parameterN: and $ret$

llStar translates an LLVM module to a coreStar CFG:

\[
x = \text{add} \ t \ v_1, \ v_2 \leadsto x = \{\text{emp}\}\{\text{ret} = \text{bvadd}(v_1, v_2)\} \\
\text{unreachable} \leadsto \{\text{false}\}\{\text{false}\} \\
x = \text{load} \ t* \ v \leadsto x = \{v \mapsto e\}\{\text{ret} = e * v \overset{t}{\mapsto} e\}
\]
From LLVM-IR to coreStar

coreStar input language

\[
CI ::= \text{label}(l) \quad \text{block label} \\
| \text{goto}(l_1, \ldots, l_n) \quad \text{non-deterministic jump} \\
| x = \{A\}\{B\}(e_0, \ldots, e_N) \quad \text{Hoare triple}
\]

\((A, B)\) separation logic formulas with free variables

\(@\text{parameter}_0: , \ldots, @\text{parameter}_N: \) and \$\text{ret}\$

llStar translates an LLVM module to a coreStar CFG:

\[
x = \text{add} \ t \ v_1, v_2 \rightsquigarrow x = \{\text{emp}\}\{\text{ret} = \text{bvadd}(v_1, v_2)\}
\]

unreachable \rightsquigarrow \{\text{false}\}\{\text{false}\}

\[
x = \text{load} \ t^* \ v \rightsquigarrow x = \{\text{v} \leftarrow e\}\{\text{ret} = e \ast v \leftarrow e\}
\]

\[
\text{store} \ t^* \ v_1 \ v_2 \rightsquigarrow \{v_1 \leftarrow -\}\{v_1 \leftarrow v_2\}
\]
### From LLVM-IR to coreStar

**coreStar input language**

\[
CI ::= \text{label}(l) \quad \text{block label}
\]

\[
| \text{goto}(l_1, \ldots, l_n) \quad \text{non-deterministic jump}
\]

\[
| x = \{A\}\{B\}(e_0, \ldots, e_N) \quad \text{Hoare triple}
\]

(A, B separation logic formulas with free variables
@parameter0:, ..., @parameterN: and $ret)

llStar translates an LLVM module to a coreStar CFG:

\[
x = \text{add} \ t \ v_1, v_2 \leadsto x = \{\text{emp}\}\{\text{ret} = \text{bvadd}(v_1, v_2)\}
\]

unreachable $\leadsto \{\text{false}\}\{\text{false}\}$

\[
x = \text{load} \ t* \ v \leadsto x = \{v \mapsto e\}\{\text{ret} = e * v \mapsto e\}
\]

store \(t* \ v_1 \ v_2 \leadsto \{v_1 \mapsto -\}\{v_1 \mapsto v_2\}
\]

\[
x = \text{alloca}(t) \leadsto x = \{\text{emp}\}\{\text{ret} \mapsto x'\}
\]
From LLVM-IR to coreStar

coreStar input language

\[
CI ::= \text{label}(l) \quad \text{block label}
\]

\[
\mid \text{goto}(l_1, \ldots, l_n) \quad \text{non-deterministic jump}
\]

\[
\mid x = \{A\}\{B\}(e_0, \ldots, e_N) \quad \text{Hoare triple}
\]

\[
(A, B \text{ separation logic formulas with free variables}
\]

\[
@\text{parameter}_0: \ldots, @\text{parameter}_N: \text{ and } \$\text{ret}
\]

llStar translates an LLVM module to a coreStar CFG:

\[
x = \text{add} \ t \ v_1, \ v_2 \leadsto x = \{\text{emp}\}\{\$\text{ret} = \text{bvadd}(v_1, v_2)\}
\]

unreachable \leadsto \{\text{false}\}\{\text{false}\}

\[
x = \text{load} \ t^* \ v \leadsto x = \{v \mapsto e\}\{\$\text{ret} = e \times v \mapsto e\}
\]

store \ t^* \ v_1, v_2 \leadsto \{v_1 \mapsto -\}\{v_1 \mapsto v_2\}

\[
x = \text{alloca}(t) \leadsto x = \{\text{emp}\}\{\$\text{ret} \mapsto x'\}
\]

\[
x = \text{call} \ t \ f(v_1, \ldots, v_n) \leadsto x = \{A\}\{B\}(v_1, \ldots, v_n)
\]
coreStar: frame inference

- To execute: $x = \{A\}\{B\}(e_1, \ldots, e_n)$
To execute: $x = \{A\}\{B\}(e_1, \ldots, e_n)$

- Current symbolic state: $C$

**coreStar: frame inference**
coreStar: frame inference

- To execute: \( x = \{A\}\{B\}(e_1, \ldots, e_n) \)
- Current symbolic state: \( C \)
- \( \Rightarrow \) Frame problem: is there \( F \) such that:

\[
C \vdash A[p_1, \ldots, p_n \leftarrow e_1, \ldots, e_n] \ast F
\]
To execute: \( x = \{A\} \{B\}(e_1, \ldots, e_n) \)

Current symbolic state: \( C \)

\( \Rightarrow \textbf{Frame problem}: \) is there \( F \) such that:

\[
C \vdash A[p_1, \ldots, p_n \leftarrow e_1, \ldots, e_n] \ast F
\]

If \( F \) exists, then the resulting state is (\( x' \) fresh):

\[
(B[p_1, \ldots, p_n \leftarrow e_1, \ldots, e_n] \ast F)[x \leftarrow x'][\text{ret} \leftarrow x]
\]
coreStar: frame inference

- To execute: \( x = \{ A \} \{ B \}(e_1, \ldots, e_n) \)
- Current symbolic state: \( C \)
- \( \Rightarrow \) **Frame problem**: is there \( F \) such that:
  \[
  C \vdash A[p_1, \ldots, p_n \leftarrow e_1, \ldots, e_n] * F
  \]
- If \( F \) exists, then the resulting state is (\( x' \) fresh):
  \[
  (B[p_1, \ldots, p_n \leftarrow e_1, \ldots, e_n] * F)[x \leftarrow x'][\text{ret} \leftarrow x]
  \]
- Problem: coreStar doesn’t know how to reason about our **spatial predicates**!
coreStar: frame inference

- To execute: \( x = \{A\}\{B\}(e_1, \ldots, e_n) \)
- Current symbolic state: \( C \)
- ⇒ **Frame problem**: is there \( F \) such that:
  \[
  C \vdash A[p_1, \ldots, p_n \leftarrow e_1, \ldots, e_n] \ast F
  \]
- If \( F \) exists, then the resulting state is \( (x' \text{ fresh}) \):
  \[
  (B[p_1, \ldots, p_n \leftarrow e_1, \ldots, e_n] \ast F)[x \leftarrow x'][\$\text{ret} \leftarrow x]
  \]
- Problem: coreStar doesn’t know how to reason about our spatial predicates!
- Solution: coreStar expects **rewrite rules** to help solve frame problems
IlStar rules for simple pointers

- Rules transform the current goal $A \vdash B$ to hypotheses $A_1 \vdash B_1, \ldots, A_n \vdash B_n$.
llStar rules for simple pointers

- Rules transform the current goal $A \vdash B$ to hypotheses $A_1 \vdash B_1, \ldots, A_n \vdash B_n$.
- Implicitly rewrites $F \ast A \vdash B \ast F'$ to $F \ast A_1 \vdash B_1 \ast F', \ldots, F \ast A_n \vdash B_n \ast F'$. 

Pattern variables of the form $?x$ can be used to match any expression.
llStar rules for simple pointers

- Rules transform the current goal $A \vdash B$ to hypotheses $A_1 \vdash B_1, \ldots, A_n \vdash B_n$.
- Implicitly rewrites $F \ast A \vdash B \ast F'$ to $F \ast A_1 \vdash B_1 \ast F', \ldots, F \ast A_n \vdash B_n \ast F'$.
- Rules for pointer predicate:

$$null \mapsto ?v \vdash \quad \quad \quad \quad \quad \vdash ?v = ?w \quad \quad \quad \quad \quad \vdash ?x \mapsto ?v \vdash ?x \mapsto ?w$$

Pattern variables of the form $?x$ can be used to match any expression.
llStar rules for simple pointers

- Rules transform the current goal $A \vdash B$ to hypotheses $A_1 \vdash B_1, \ldots, A_n \vdash B_n$.
- Implicitly rewrites $F \ast A \vdash B \ast F'$ to $F \ast A_1 \vdash B_1 \ast F', \ldots, F \ast A_n \vdash B_n \ast F'$.
- Rules for pointer predicate:

\[
\begin{align*}
\text{null } \mapsto t & \quad \vdash ?v \vdash \\
?x \mapsto t & \quad \vdash ?v \vdash \quad ?x \mapsto t \\
\quad \vdash ?v = ?w &
\end{align*}
\]

- Pattern variables of the form $?x$ can be used to match any expression.
IlStar automatically generates rules for structs. For instance, for

```c
%s = type { i32, %s* }
```

\[
\text{sizeof}(%s) = 16
\]
llStar automatically generates rules for structs. For instance, for

\[ \%s = \text{type}\ \{\ \text{i32},\ \%s^*\ \} \]

\[
\text{sizeof}(\%s) = 16
\]

\[
\begin{align*}
?x & \xrightarrow{\text{i32}} \text{field}^0_{\%s}(?v) \ast \ ?x + 8 & \xrightarrow{\%s^*} & \text{field}^1_{\%s}(?v) \vdash \ ?x & \xrightarrow{\text{i32}} & ?w \\
?x & \xrightarrow{\%s} & ?v \vdash \ ?x & \xrightarrow{\text{i32}} & ?w
\end{align*}
\]
IlStar automatically generates rules for structs. For instance, for
\[ %s = \text{type} \{ \text{i32}, %s* \} \]

\[ \text{sizeof}(%s) = 16 \]

\[ \frac{?x \leftarrow \text{i32}}{?x \rightarrow \text{field}^0_{%s}(?v) \ast \ ?x + 8 \leftarrow %s* \rightarrow \text{field}^1_{%s}(?v) \mid \ ?x \leftarrow \text{i32}} \rightarrow \text{?w} \]

\[ ?x \rightarrow \text{?v} \mid \ ?x \leftarrow \text{i32} \rightarrow \text{?w} \]

\[ \frac{?x \leftarrow \text{i32}}{?x \rightarrow \text{field}^0_{%s}(?v) \ast \ ?x + 8 \leftarrow %s* \rightarrow \text{field}^1_{%s}(?v) \mid \ ?x + 8 \leftarrow %s* \rightarrow \text{?w}} \]

\[ ?x \rightarrow \text{?v} \mid \ ?x + 8 \leftarrow %s* \rightarrow \text{?w} \]
The LLVM instruction does pointer arithmetic (no heap access).

\[
x = \text{getelementptr} \ t^* \ v, \ t_1 \ v_1, \ldots, \ t_n \ v_n
\]

\[\leadsto x = \{\text{emp}\}\{\text{ret} = \text{eltptr}(v, t^*, [v_1; \ldots; v_n])\}\]
The LLVM instruction does pointer arithmetic (no heap access).

\[
x = \text{getelementptr } t* \ v, t_1 \ v_1, \ldots, t_n \ v_n \\
\leadsto x = \{\text{emp}\}{\$ret = \text{eltptr}(v, t*, [v_1; \ldots; v_n])}
\]

Reasoning rules (\(\%s = \text{type } \{ \text{i32, \%s* } \}):

\[
\text{eltptr}(?x, ?t, []) = ?x
\]
• The LLVM instruction does pointer arithmetic (no heap access).

\[
x = \text{getelementptr } t^* v, t_1 v_1, \ldots, t_n v_n \\
\leadsto x = \{\text{emp}\}\{\text{ret } eltptr(v, t^*, [v_1; \ldots; v_n])\}
\]

• Reasoning rules (%s = type \{ i32, %s* \}):

\[
\text{eltptr}(?x, ?t, []) = ?x \\
\text{eltptr}(?x, %s, 0 :: ?) = \text{eltptr}(?x, \text{i32}, ?)
\]
The LLVM instruction does pointer arithmetic (no heap access).

\[
x = \text{getelementptr } t^* v, t_1 v_1, \ldots, t_n v_n
\Rightarrow x = \{\text{emp}\}\{\text{ret} = \text{eltptr}(v, t^*, [v_1; \ldots; v_n])\}
\]

Reasoning rules (\(\%s = \text{type } \{ \text{i32, } \%s^* \}\)):

\[
\text{eltptr}(?x, ?t, []) = ?x
\]

\[
\text{eltptr}(?x, \%s, 0 :: ?j) = \text{eltptr}(?x, \text{i32}, ?j)
\]

\[
\text{eltptr}(?x, \%s, 1 :: ?j) = \text{eltptr}(?x + 8, \%s^*, ?j)
\]
• llStar does symbolic execution for LLVM-IR
Extending llStar

- llStar does symbolic execution for LLVM-IR
- Based on a flexible reasoning engine: coreStar
• llStar does symbolic execution for LLVM-IR
• Based on a flexible reasoning engine: coreStar
• llStar generates reasoning rules for LLVM datatypes
Extending llStar

- llStar does symbolic execution for LLVM-IR
- Based on a flexible reasoning engine: coreStar
- llStar generates reasoning rules for LLVM datatypes
- Easily extensible with more data structures
Extending llStar

- llStar does symbolic execution for LLVM-IR
- Based on a flexible reasoning engine: coreStar
- llStar generates reasoning rules for LLVM datatypes
- Easily extensible with more data structures
- ... Just add more reasoning rules!
Conclusion
• llStar can **verify** small/medium sequential LLVM-IR programs, and reason about their low-level features
Summary

- \texttt{llStar} can \textbf{verify} small/medium sequential LLVM-IR programs, and reason about their low-level features
- \textbf{Compositional reasoning} using separation logic: per-function, bottom-up verification, no \texttt{main()} needed, etc.
IlStar can **verify** small/medium sequential LLVM-IR programs, and reason about their low-level features.

- **Compositional reasoning** using separation logic: per-function, bottom-up verification, no `main()` needed, etc.
- **Extensible** reasoning engine enables custom theories to be quickly added and tested on real code.
llStar can verify small/medium sequential LLVM-IR programs, and reason about their low-level features

**Compositional reasoning** using separation logic: per-function, bottom-up verification, no `main()` needed, etc.

**Extensible** reasoning engine enables custom theories to be quickly added and tested on real code

**Cool(?) Web interface**
IlStar can **verify** small/medium sequential LLVM-IR programs, and reason about their low-level features.

**Compositional reasoning** using separation logic: per-function, bottom-up verification, no `main()` needed, etc.

**Extensible** reasoning engine enables custom theories to be quickly added and tested on real code.

**Cool(?) Web interface**

Extended the coreStar tool in the process.
Future work

- Concurrency
Future work

- Concurrency
- Nicer syntax for formulas
Future work

- Concurrency
- Nicer syntax for formulas
- Better error reports
Future work

- Concurrency
- Nicer syntax for formulas
- Better error reports
- Make specs of LLVM-IR commands and rule generation also parameterisable
Future work

- Concurrency
- Nicer syntax for formulas
- Better error reports
- Make specs of LLVM-IR commands and rule generation also parameterisable
- Inference of pre/post-conditions using biabduction
  ⇒ prove large programs
Future work

- Concurrency
- Nicer syntax for formulas
- Better error reports
- Make specs of LLVM-IR commands and rule generation also parameterisable
- Inference of pre/post-conditions using biabduction ➞ prove large programs
- Connect to mechanised semantics (e.g. Vellvm)
Future work

- Concurrency
- Nicer syntax for formulas
- Better error reports
- Make specs of LLVM-IR commands and rule generation also parameterisable
- Inference of pre/post-conditions using biabduction \( \Rightarrow \) prove large programs
- Connect to mechanised semantics (e.g. Vellvm)
- Ongoing collaboration to rewrite coreStar entirely with Grigore (Oxford), Peterson (MSR), Tzevelekos (Queen Mary), Gorogiannis (Middlesex)
Here be Wyverns!
Verifying LLVM-IR with llStar

Jules Villard

Imperial College London

http://llstar.pauvre.org/