On Formalising Predicated Execution and Predicate-Aware Scheduling

Yann Herklotz and John Wickerson

Imperial College London

What is High-Level Synthesis?

High-Level Synthesis (HLS)

Conversion from an algorithmic, sequential description in C to a parallel hardware design in Verilog.

Naïve Implementation

Fixing Symbolic Representation

On Predicate Evaluation

Results and Conclusion

What is High-Level Synthesis?

High-Level Synthesis (HLS)

Conversion from an algorithmic, sequential description in C to a parallel hardware design in Verilog.

Unreliability of HLS

We found that HLS tools had incorrect output for 1.5% of simple, random C code.[1]

[1] Yann Herklotz, Zewei Du, Nadesh Ramanathan, and John Wickerson. *An empirical study of the reliability of high-level synthesis tools*. In 29th IEEE Annual Int. Symp. on FCCM, 2021.

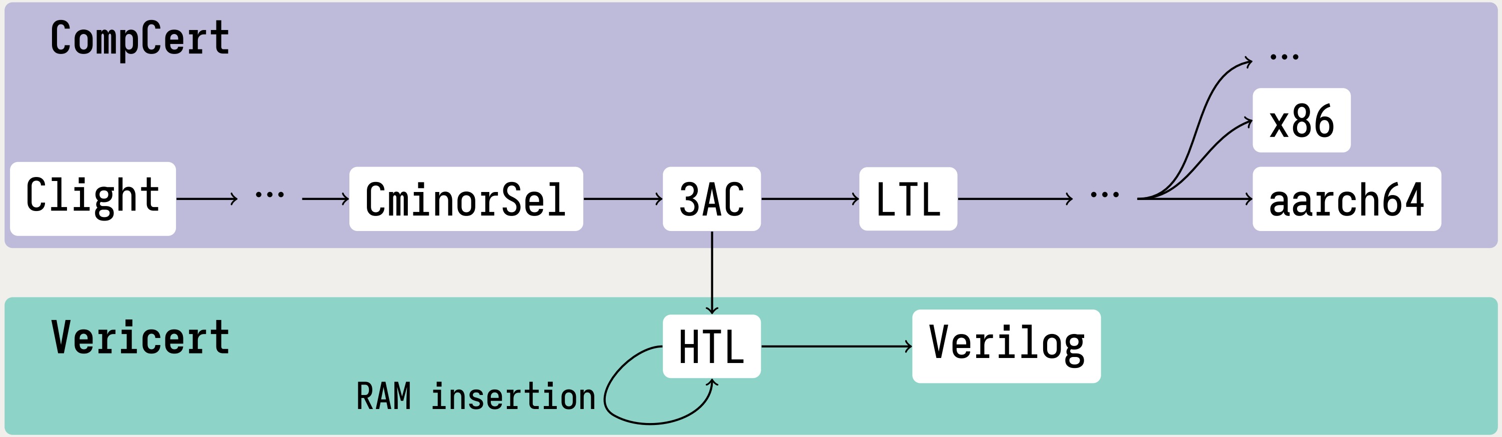
Naïve Implementation

Fixing Symbolic Representation

On Predicate Evaluation

Results and Conclusion

Solution: Formally Verified HLS



- Build a verified HLS tool on top of CompCert called Vericert.
- Currently only generates sequential hardware.

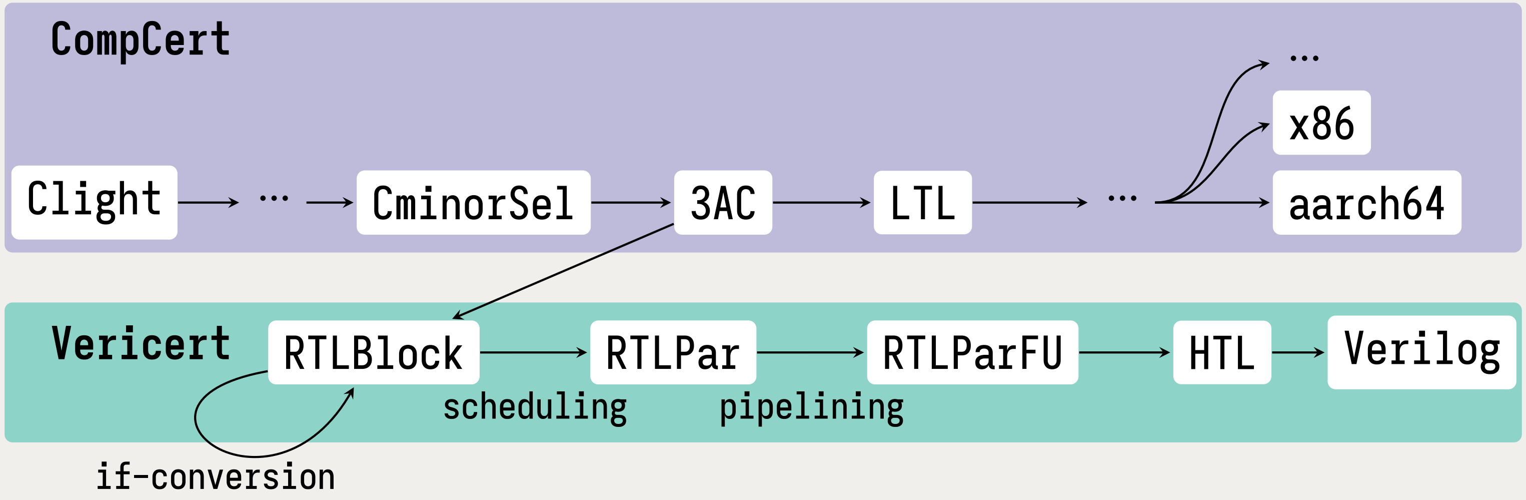
Naïve Implementation

Fixing Symbolic Representation

On Predicate Evaluation

Results and Conclusion

Adding Instruction-Level Parallelism



Add instruction level parallelism using predicated instructions in basic blocks.

Example of Instruction Scheduling

```
r2 = r1 + r4;
if p1: r1 = r2 + r4;
if !p1&&!p2: r3 = r1 * r1;
if p1: p3 = r2 == r3;
```

```
r2 = r1 + r4
  || if !p1&&!p2: r3 = r1 * r1;
if p1: r1 = r2 + r4;
  || if (p1) p3 = r2 == r3;
```


Naïve Implementation

Fixing Symbolic Representation

On Predicate Evaluation

Results and Conclusion

Naïve Implementation

Naïve Implementation

Fixing Symbolic Representation

On Predicate Evaluation

Results and Conclusion

RTLBlock and RTLPar Syntax

$$\mathcal{B} ::= \text{slist } \mathcal{I}$$

$$\mathcal{P} ::= \text{slist (plist (slist } \mathcal{I}\text{))}$$

Naïve Implementation

Fixing Symbolic Representation

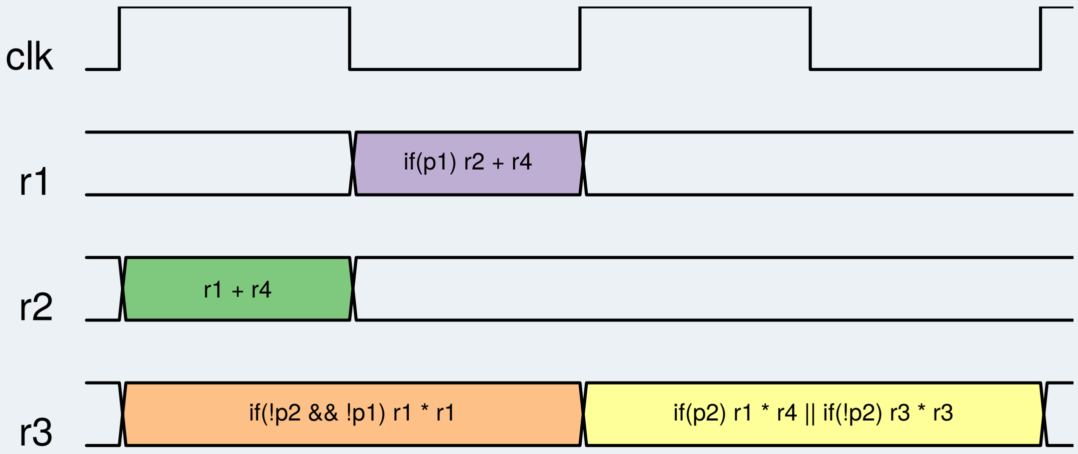
On Predicate Evaluation

Results and Conclusion

RTLBlock and RTLPar Syntax

$\mathscr{B}$ ::= slist $\mathscr{I}$

$\mathscr{P}$ ::= slist (plist (slist $\mathscr{I}$))



Naïve Implementation

Fixing Symbolic Representation

On Predicate Evaluation

Results and Conclusion

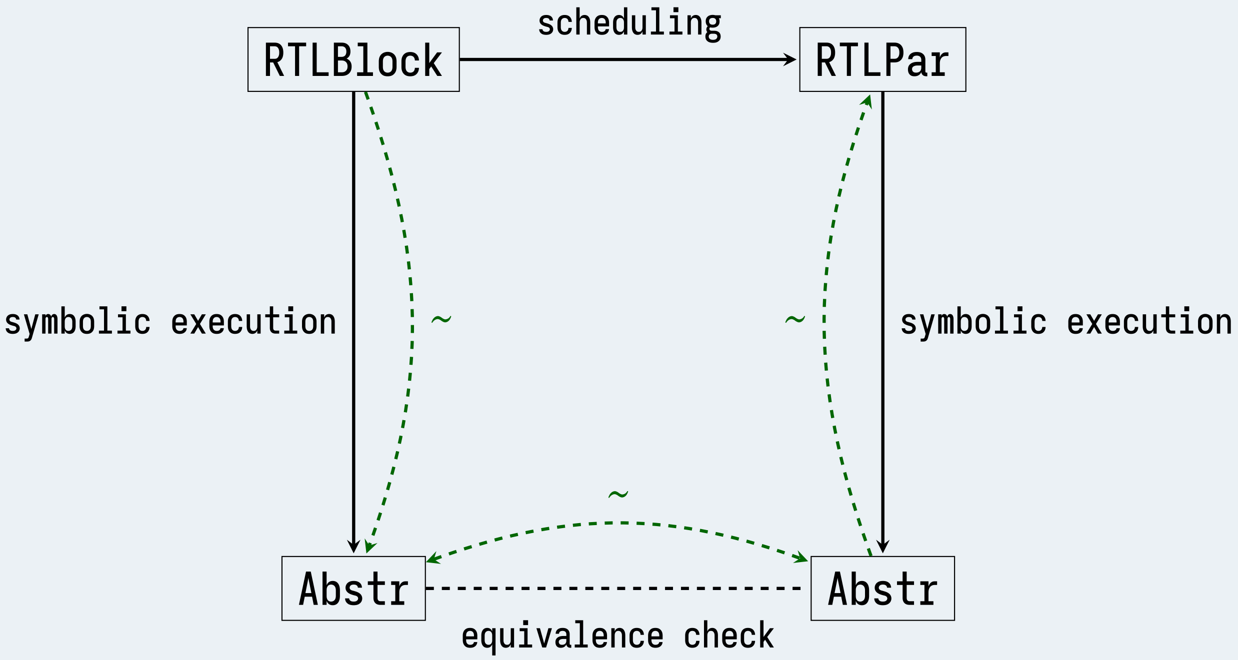
RTLBlock and RTLPar Syntax

$\mathscr{B}$ ::= slist $\mathscr{I}$

$\mathscr{P}$ ::= slist (plist (slist $\mathscr{I}$))

$$
\begin{array}{rcl}
\mathscr{I} & ::= & \texttt{nop} \\
 & | & \texttt{if P: } r = r + r \;|\; \cdots \\
 & | & \texttt{if P: } r = M[a] \\
 & | & \texttt{if P: } M[a] = r \\
 & | & \texttt{if P: } p = r == r \;|\; \cdots \\
 & | & \texttt{if P: exit } \mathscr{C}
\end{array}
$$

Translation Validation in CompCert



Checker Implementation

- Assuming we have sequential and parallel code we want to compare.
- **Symbolically execute** the sequential and parallel code.
- Describe an **equivalence checker** for the results of symbolic execution.

$$R \texttt{ : } r \mid p \mid \texttt{M} \mid \texttt{Exit}$$

$$\Sigma \texttt{ : } R \texttt{ -> } (R \texttt{ -> val}) \texttt{ -> val}$$

$$\texttt{SExec : } \mathscr{B} \texttt{ -> } \Sigma \qquad \texttt{check : } \Sigma \texttt{ -> } \Sigma \texttt{ -> bool}$$

Naïve Implementation

Fixing Symbolic Representation

On Predicate Evaluation

Results and Conclusion

Example Execution

```
r2 = r1 + r4;
if p1: r1 = r2 + r4;
if !p1&&!p2: r3 = r1 * r1;
if p1: p3 = r2 == r3;
```

$$r1 \longmapsto \begin{array}{l}(p1_0 \longrightarrow (\mathtt{r1}_0\mathtt{+r4}_0)\mathtt{+r4}_0) \\ \wedge\ (\neg p1_0 \longrightarrow \mathtt{r1}_0)\end{array}$$

$$r2 \longmapsto \mathtt{r1}_0\mathtt{+r4}_0$$

$$r3 \longmapsto \begin{array}{l}(\neg p1_0 \wedge \neg p2_0) \longrightarrow \\ \quad \left(\begin{array}{l}(p1_0 \longrightarrow (\mathtt{r1}_0\mathtt{+r4}_0)\mathtt{+r4}_0) \\ \wedge\ (\neg p1_0 \longrightarrow \mathtt{r1}_0)\end{array}\right) \mathtt{*}\ \ \ldots) \\ \wedge\ ((p1_0 \vee p2_0) \longrightarrow \mathtt{r3}_0)\end{array}$$

$$p3 \longmapsto \begin{array}{l}(\neg p1_0 \longrightarrow p3_0) \\ \wedge\ (p1_0 \longrightarrow (\mathtt{r1}_0\mathtt{+r4}_0\ \ \mathtt{==}\ \ \ldots))\end{array}$$

A Few Problems Arise

- Very **recursive** structure of guarded expressions.
- Representation is similar to SMT **formulas** with **atoms**.
- Currently **atoms** can contain **formulas** too.

$$P \ ::= \ \mathrm{p}_0 \ | \ P \vee P \ | \ P \ \mathbf{+} \ P \ | \ P \ \mathbf{==} \ P \ | \ \cdots$$

Fixing Symbolic Representation

Naïve Implementation

Fixing Symbolic Representation

On Predicate Evaluation

Results and Conclusion

Execution With Flatter Representation

```
r2 = r1 + r4;
if p1: r1 = r2 + r4;
if !p1&&!p2: r3 = r1 * r1;
if p1: p3 = r2 == r3;
```

$$r1 \longmapsto \begin{cases} (r1_0 + r4_0) + r4_0, & \text{if } p1_0 \\ r1_0, & \text{if } \neg p1_0 \end{cases}$$

$$r2 \longmapsto r1_0 + r4_0$$

$$r3 \longmapsto \begin{cases} r1_0 * r1_0, & \text{if } \neg p1_0 \land \neg p1_0 \land \neg p2_0 \\ \dots \end{cases}$$

$$p3 \longmapsto \begin{array}{l} (\neg p1_0 \longrightarrow p3_0) \\ \land\ ((p1_0 \lor p2_0) \longrightarrow ((r1_0 + r4_0) + r4_0\ == \ r3_0)) \end{array}$$

Naïve Implementation

Fixing Symbolic Representation

On Predicate Evaluation

Results and Conclusion

As a Grammar

$$G \texttt{ ::= } [(P,\ e)]$$

$$P \texttt{ ::= } \mathtt{p}_0 \mid P \vee P \mid e \texttt{ == } e \mid \cdots$$

$$e \texttt{ ::= } \mathtt{r}_0 \mid e \texttt{ + } e \mid e \texttt{ * } e \mid e[e] \mid \cdots$$

$$F \texttt{ ::= } r \longmapsto G \texttt{ ; } \mathtt{M} \longmapsto G \texttt{ ; } p \longmapsto P \texttt{ ; } \mathtt{Exit} \longmapsto [(P,\ \mathscr{C})]$$

Naïve Implementation

Fixing Symbolic Representation

On Predicate Evaluation

Results and Conclusion

Defining the Equivalence Check

We have syntactic equality for expressions implying same behaviour:

$$\frac{(e, \sigma_{\mathscr{B}}) \Downarrow v \qquad \sigma_{\mathscr{B}} \sim \sigma_{\mathscr{P}}}{(e, \sigma_{\mathscr{P}}) \Downarrow v}$$

Now to compare guarded expressions $G_{\mathscr{B}} = [(P_{\mathscr{B}}, e_{\mathscr{B}}), \cdots]$ and $G_{\mathscr{P}} = [(P_{\mathscr{P}}, e_{\mathscr{P}}), \cdots]$, we can use a verified SAT solver:

$$\begin{gathered}(P_{\mathscr{B}} \longrightarrow e_{\mathscr{B}} \land \cdots) \\ \Longleftrightarrow \\ (P_{\mathscr{P}} \longrightarrow e_{\mathscr{P}} \land \cdots)\end{gathered}$$

Naïve Implementation

Fixing Symbolic Representation

On Predicate Evaluation

Results and Conclusion

On Predicate Evaluation

Predicate Evaluation Can Block

$$\bot \iff (\bot \land \texttt{x == y})$$

- SAT solver will say equivalent.
- `x == y` can block and therefore might not behave the same.
- For example when doing pointer equality with **invalid** pointers.
- This requires us to define a **well-formedness** condition for predicates.

Naïve Implementation

Fixing Symbolic Representation

On Predicate Evaluation

Results and Conclusion

Well-Formedness of Predicates

$$P_1 \iff P_2$$

- Check that predicate from the output of the schedule only contains **executable atoms**.
- $\alpha(P)$ retrieves the atoms of P.

$$\alpha(P_1) \supseteq \alpha(P_2)$$

P_2 will be executable if P_1 is executable as well.

Naïve Implementation

Fixing Symbolic Representation

On Predicate Evaluation

Results and Conclusion

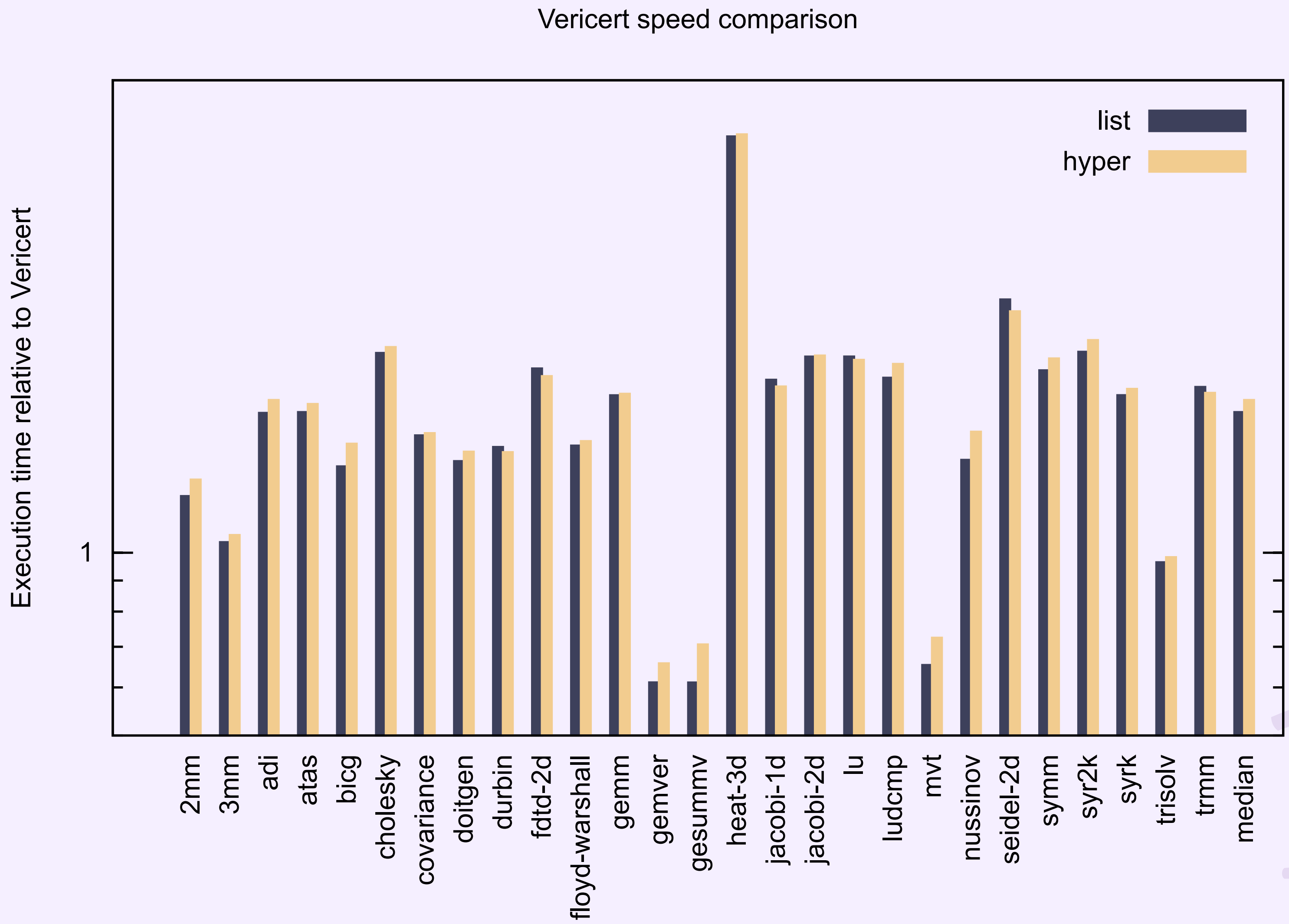
Results and Conclusion

Naïve Implementation

Fixing Symbolic Representation

On Predicate Evaluation

Results and Conclusion


Vericert speed comparison
Execution time relative to Vericert
1
list
hyper
2mm
3mm
adi
atas
bicg
cholesky
covariance
doitgen
durbin
fdtd-2d
floyd-warshall
gemm
gemver
gesummv
heat-3d
jacobi-1d
jacobi-2d
lu
ludcmp
mvt
nussinov
seidel-2d
symm
syr2k
syrk
trisolv
trmm
median

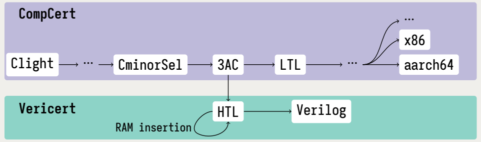
Conclusion

- SAT solver can be used to write a translation validation pass in CompCert and to help prove the forward simulation.
- Performance is ~1.8× better than base Vericert, now around 2× slower than optimised LegUp.
- Verified most passes (if-conversion, basic block generation, symbolic execution soundness).
- Currently finishing equivalence checking proof.

github.com/ymherklotz/vericert

Thank you!

Solution: Formally Verified HLS



- Build a verified HLS tool on top of CompCert.
- Currently only generates sequential hardware.

A Few Problems Arise

- Very recursive structure of guarded expressions.
- Representation is similar to SMT formulas with atoms.
- Currently atoms can contain formulas too.

$$P ::= \mathrm{p}_0 \mid P \lor P \mid P + P \mid P == P \mid \cdots$$

Defining the Equivalence Check

We have syntactic equality for expressions implying same behaviour:

$$\frac{(e,\sigma_{\mathcal{B}}) \Downarrow v \qquad \sigma_{\mathcal{B}} \sim \sigma_{\mathcal{P}}}{(e,\sigma_{\mathcal{P}}) \Downarrow v}$$

Now to compare guarded expressions $G_{\mathcal{B}} = [(P_{\mathcal{B}}, e_{\mathcal{B}}), \cdots]$ and $G_{\mathcal{P}} = [(P_{\mathcal{P}}, e_{\mathcal{P}}), \cdots]$, we can use a verified SAT solver:

$$(P_{\mathcal{B}} \longrightarrow e_{\mathcal{B}} \land \cdots) \iff (P_{\mathcal{P}} \longrightarrow e_{\mathcal{P}} \land \cdots)$$

Predicate Evaluation Can Block

$$\perp \iff (\perp \land x == y)$$

- SAT solver will say equivalent.
- x == y can block and therefore might not behave the same.
- For example when doing pointer equality with invalid pointers.
- This requires us to define a well-formedness condition for predicates.