Bit-precise Reasoning
via
Int-Blasting

Yoni Zohar

Joint work with:
Ahmed Irfan, Makai Mann, Aina Niemetz, Andres Nötzli, Mathias Preiner, Andrew Reynolds, Clark Barrett, Cesare Tinelli

Stanford University

The University of Iowa

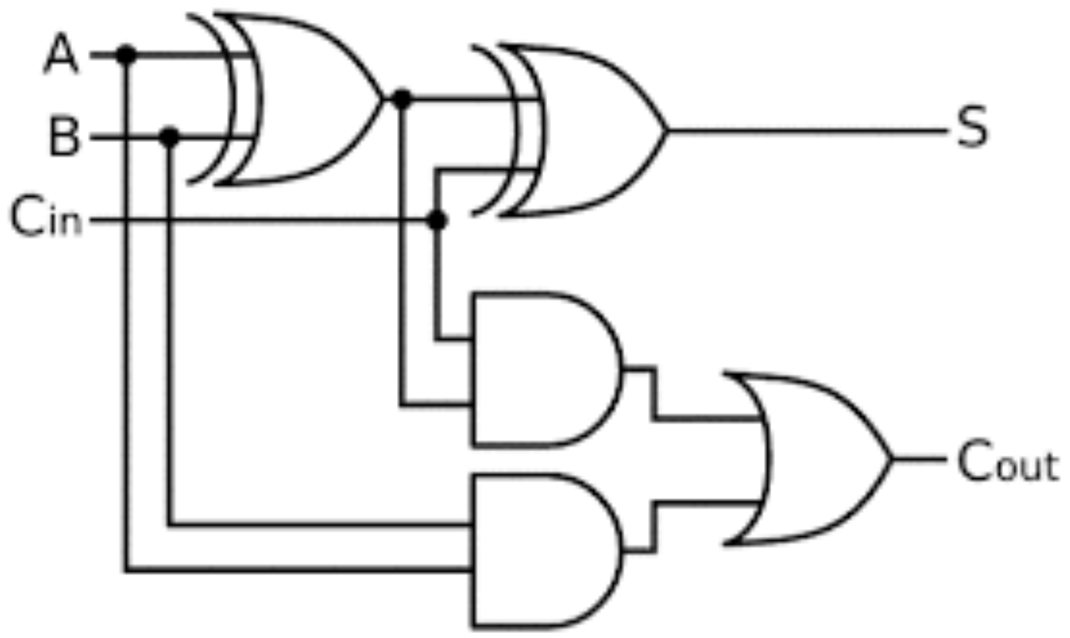
Bar-Ilan University

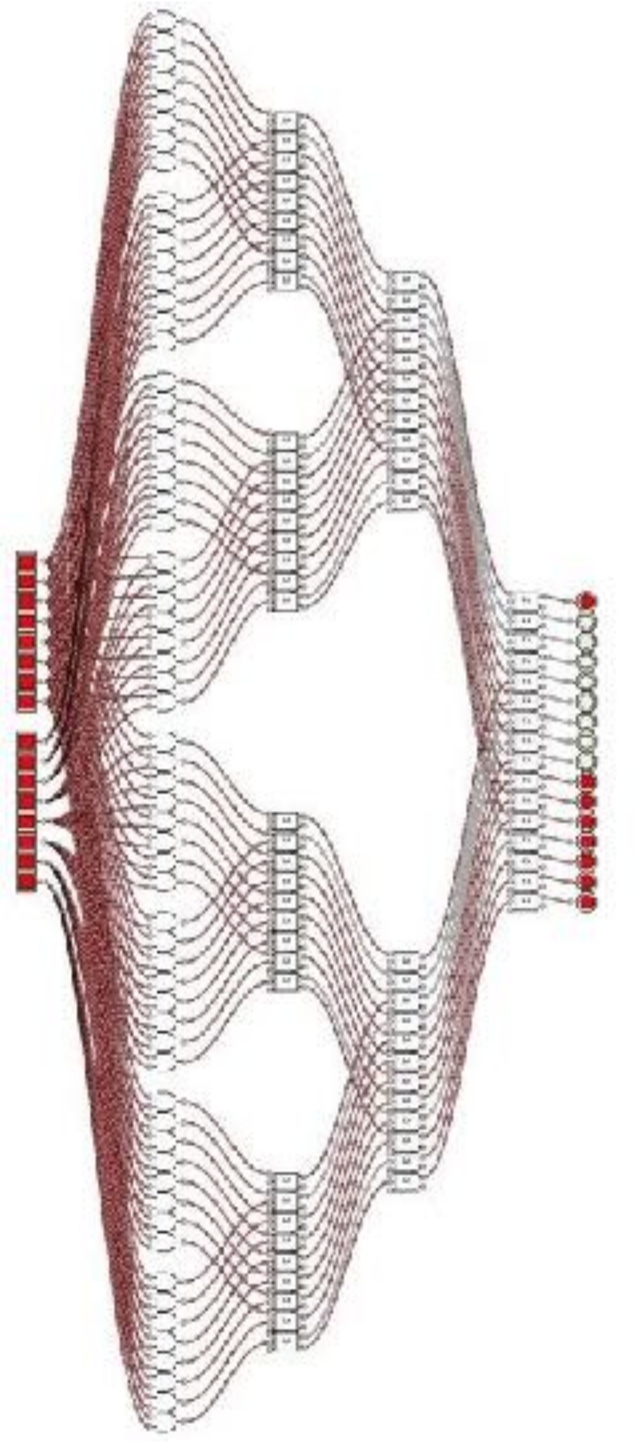
Bit-precise Reasoning

- Variables: x,y,z,…
- Constants: 0000, 01000010, 11111111, …
- Relations: =, bvult (unsigned), bvslt (signed), …
- BV Operators: bvadd (+), bvmul (·), bvand (&), bvshl (<<), …
- Logical Operators: $\wedge$, $\vee$, $\neg$, $\forall$, …
- All terms are **sorted**: BV[1], BV[2], …

Bit-vector Solving in SMT

- Bit-blasting (state-of-the-art)
  - bits — Boolean variables
  - operators — circuits
  - Scalability problems:
    - Large bit-widths (e.g., 256)
    - “Normal” bit-widths (e.g., 32) with multiplication/division
- MC-SAT
- Local Search
- **Integer approaches**





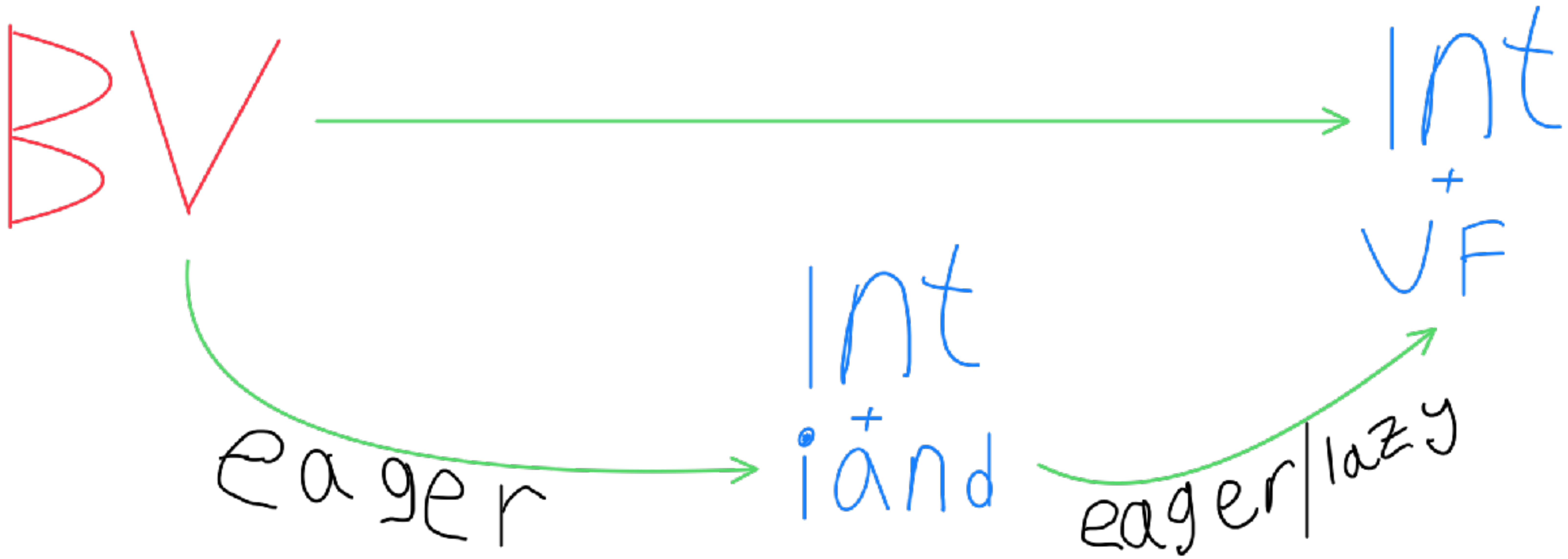
$$x + y \ \% \ 2^k$$

Integers: Inside or Outside?

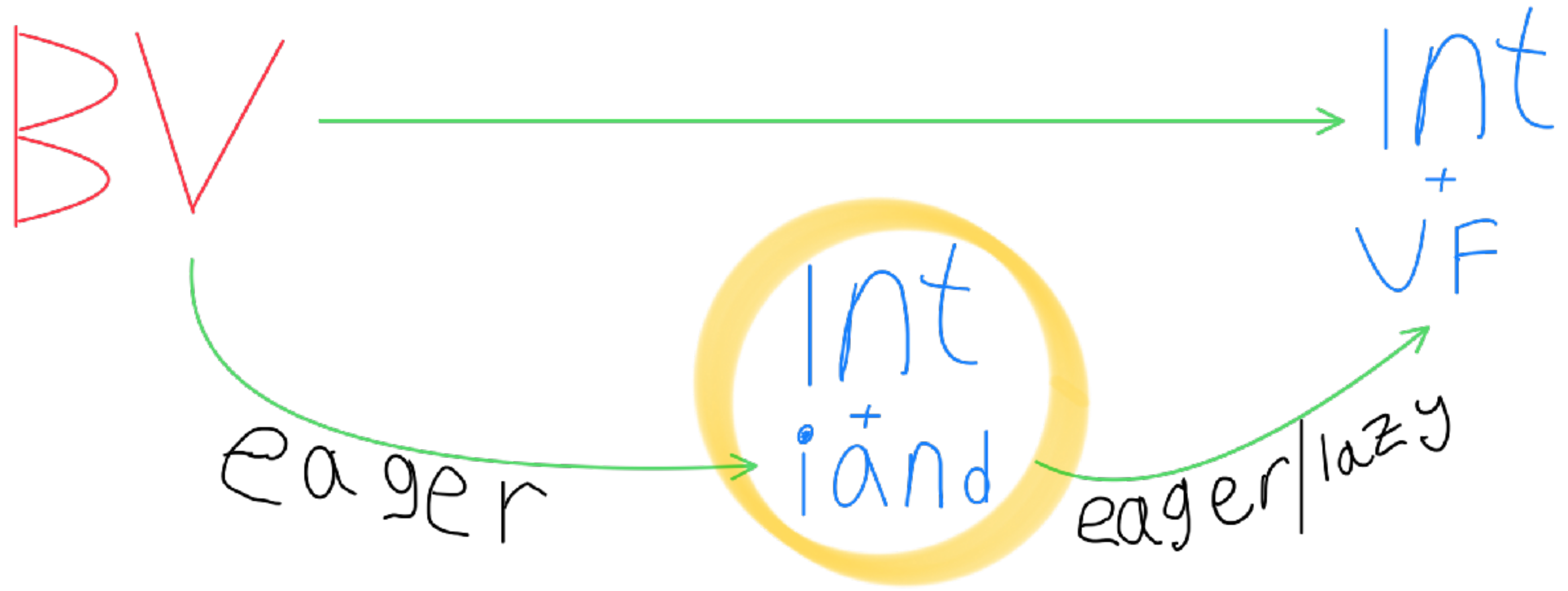
IN-N-OUT BURGER

Application Level	**Solver Level**
Eager	**Flexible**
Abstract bit-wise operations	**Abstract/refine bit-wise operations**
Black box	**More control**
Application-specific	**General**

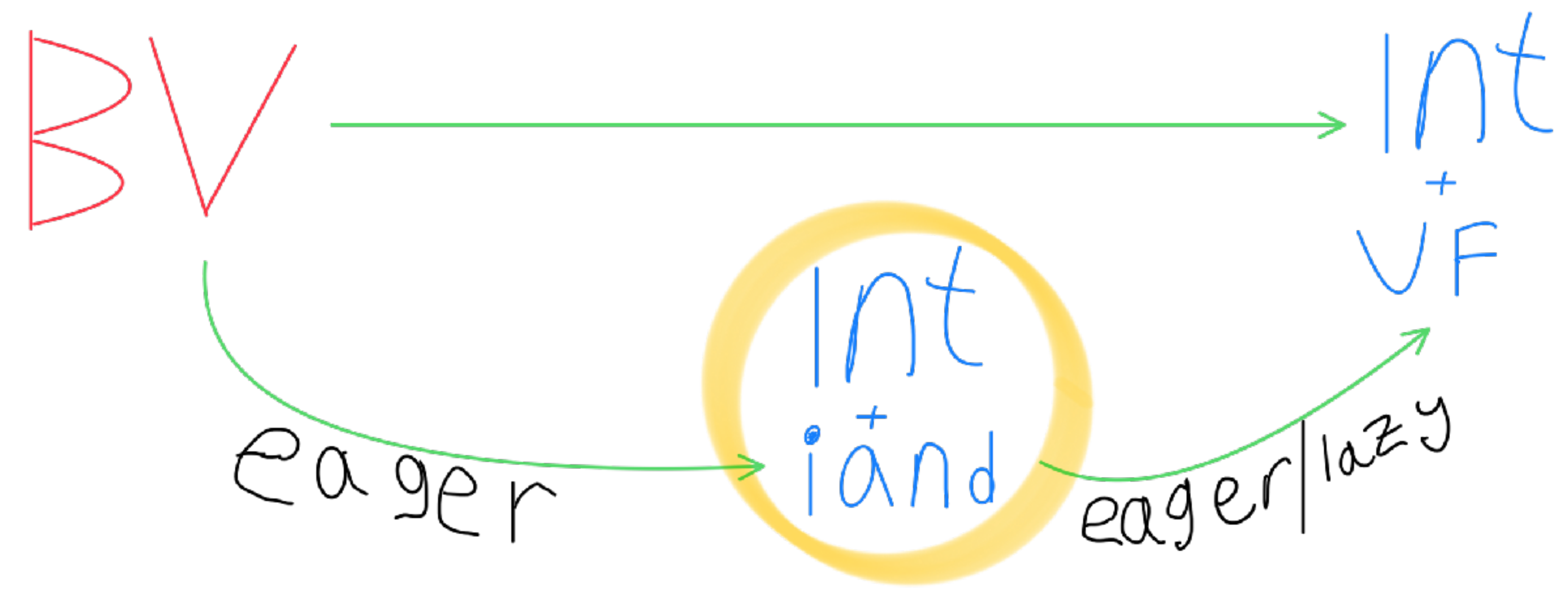
Int-blasting



Int-blasting



Arith + iand

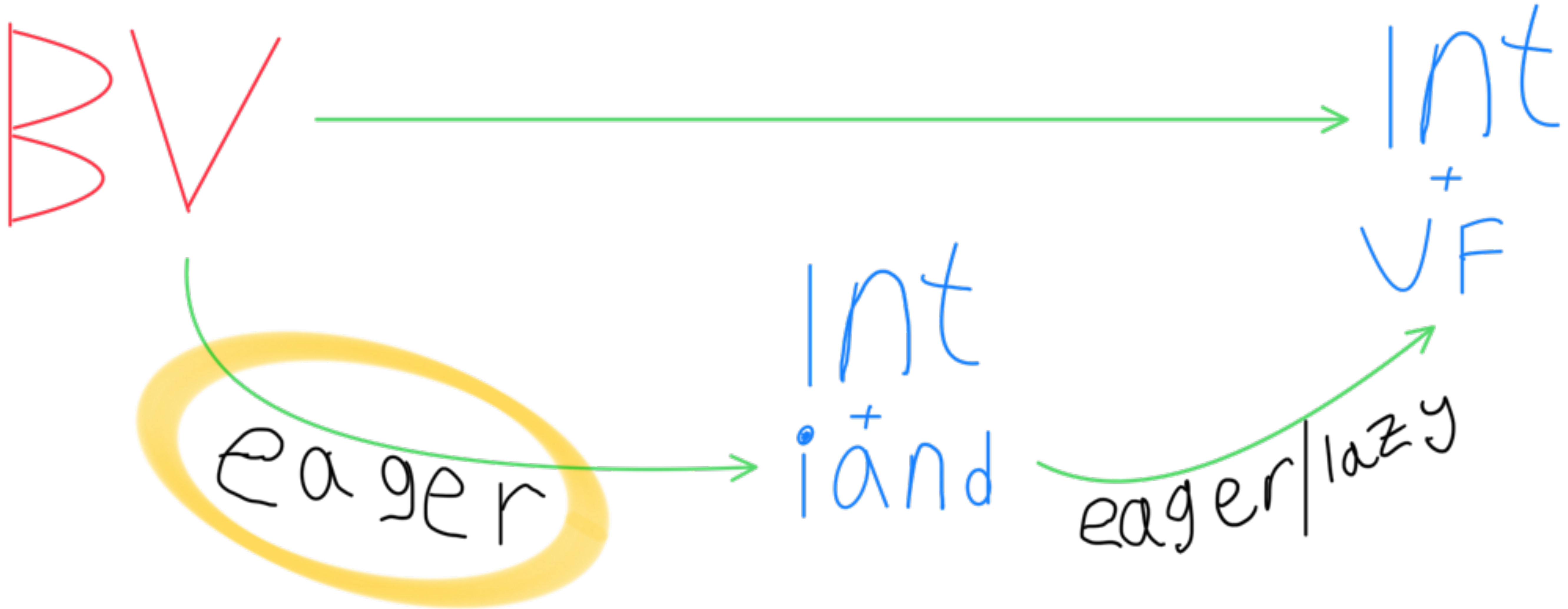


- Non-linear integer arithmetic
- iand
  - countably many binary operators $\&_1, \&_2, \ldots$
  - semantics of bit-wise “and”

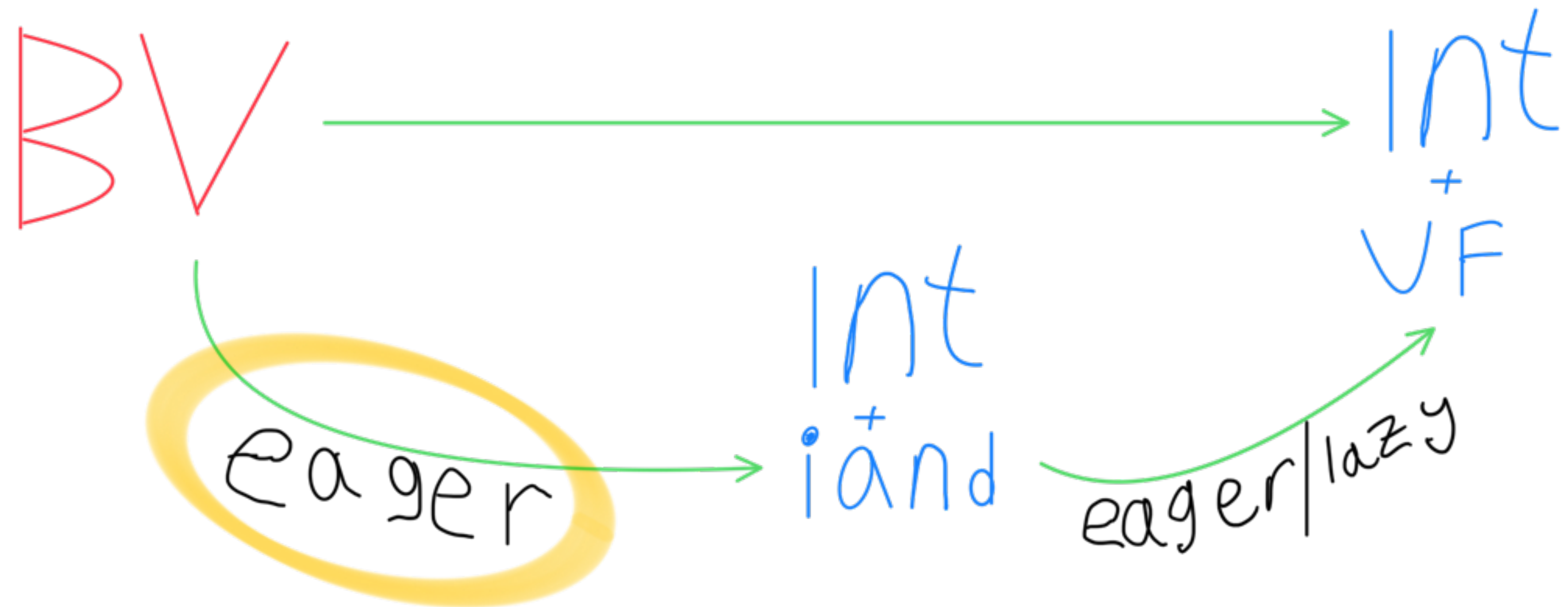
$\&_4(1,2) = 0$

$$\begin{array}{r} 0001 \\ 0010 \\ \hline 0000 \end{array}$$

Int-blasting



BV ⟶ Arith + iand



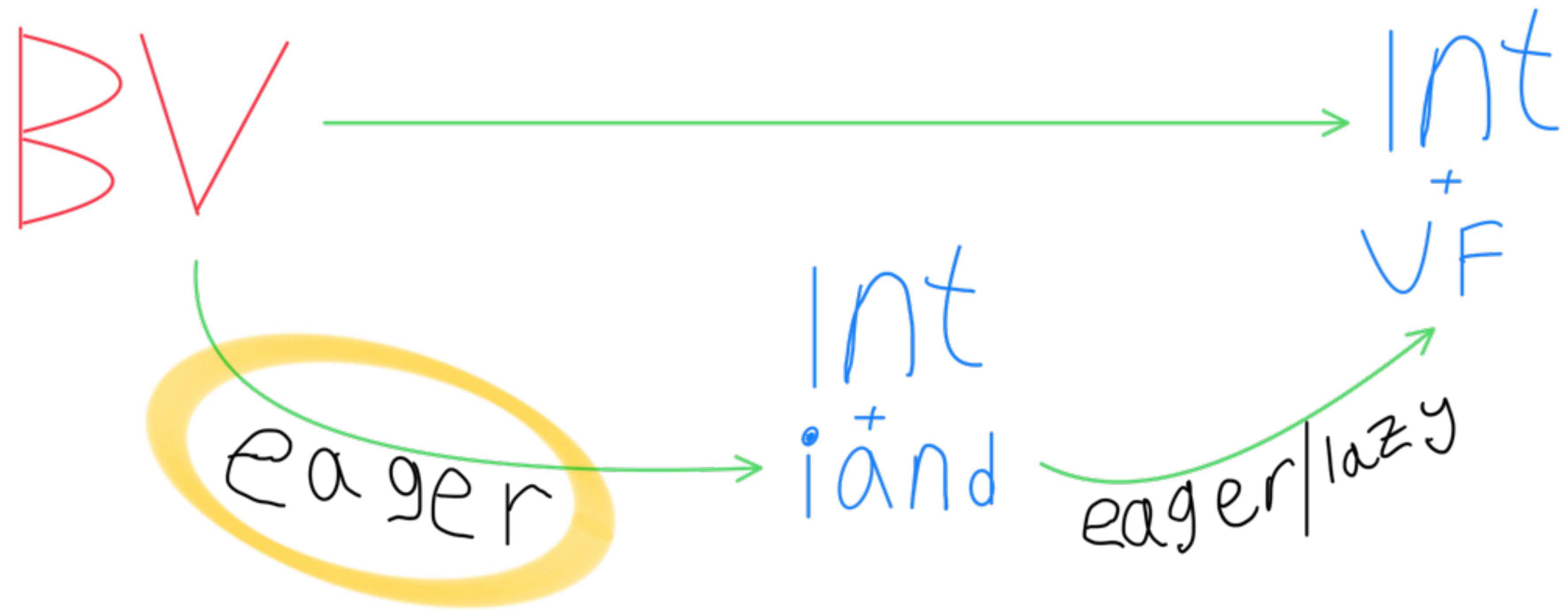
- BV variables -> Integer variables
- BV constants -> **unsigned** integer constants
- BV operators -> integer terms, based on **SMT-LIB**
- BV bit-wise “and” -> iand
- Some operators are eliminated

```
theory FixedSizeBitVectors

:smt-lib-version 2.6
:smt-lib-release "2017-11-24"
:written-by "Silvio Ranise, Cesare Tinelli, and Clark Barrett"
:date "2010-05-02"
:last-updated "2017-06-13"
:update-history
"Note: history only accounts for content changes, not release c
 2020-05-20 Fixed minor typo
 2017-06-13 Added :left-assoc attribute to bvand, bvor, bvadd,
 2017-05-03 Updated to version 2.6; changed semantics of divisi
            remainder operators.
 2016-04-20 Minor formatting of notes fields.
 2015-04-25 Updated to Version 2.5.
 2013-06-24 Renamed theory's name from Fixed_Size_Bit_Vectors t
            for consistency.
            Added :value attribute.
"
```

$$\boxed{x +_{\mathbb{BV}} y \Rightarrow x' +_{\mathbb{N}} y' \mod 2^k}$$

BV $\longrightarrow$ Arith + iand



$\underline{\mathcal{T}\ \varphi}$:

$\mathcal{C}\ \varphi \wedge \textsc{Lem}^{\leq}(\varphi)$

$\underline{\mathcal{C}\ e}$:

Match e:

$$\begin{array}{lcl} x & \rightarrow & \chi(x) \\ c & \rightarrow & [c]_{\mathbb{N}} \\ t_1 = t_2 & \rightarrow & \mathcal{C}\ t_1 = \mathcal{C}\ t_2 \end{array}$$

χ is a 1-1 mapping between BV variables and integer variables

$[\,\cdot\,]_{\mathbb{N}}$ translates bit-vectors to unsigned integers

$\textbf{BV} \longrightarrow$ Arith + iand

$$\begin{tikzcd} \mathrm{BV} \arrow[rr] \arrow[d, "\text{eager}"'] & & \mathrm{Int} + \mathrm{UF} \\ \mathrm{Int} + \mathrm{iand} \arrow[urr, "\text{eager}|\text{lazy}"'] & & \end{tikzcd}$$

$\underline{\mathcal{T}\,\varphi}$:
$\mathcal{C}\,\varphi \wedge \textsc{Lem}^{\leq}(\varphi)$

$\underline{\mathcal{C}\,e}$:
Match e:

$$\begin{array}{lcl} t_1 \bowtie^{\mathrm{BV}} t_2 & \to & \mathcal{C}\,t_1 \bowtie \mathcal{C}\,t_2 \\ t_1 {\bowtie_s}^{\mathrm{BV}} t_2 & \to & \mathsf{uts}_k(\mathcal{C}\,t_1) \bowtie \mathsf{uts}_k(\mathcal{C}\,t_2) \end{array}$$

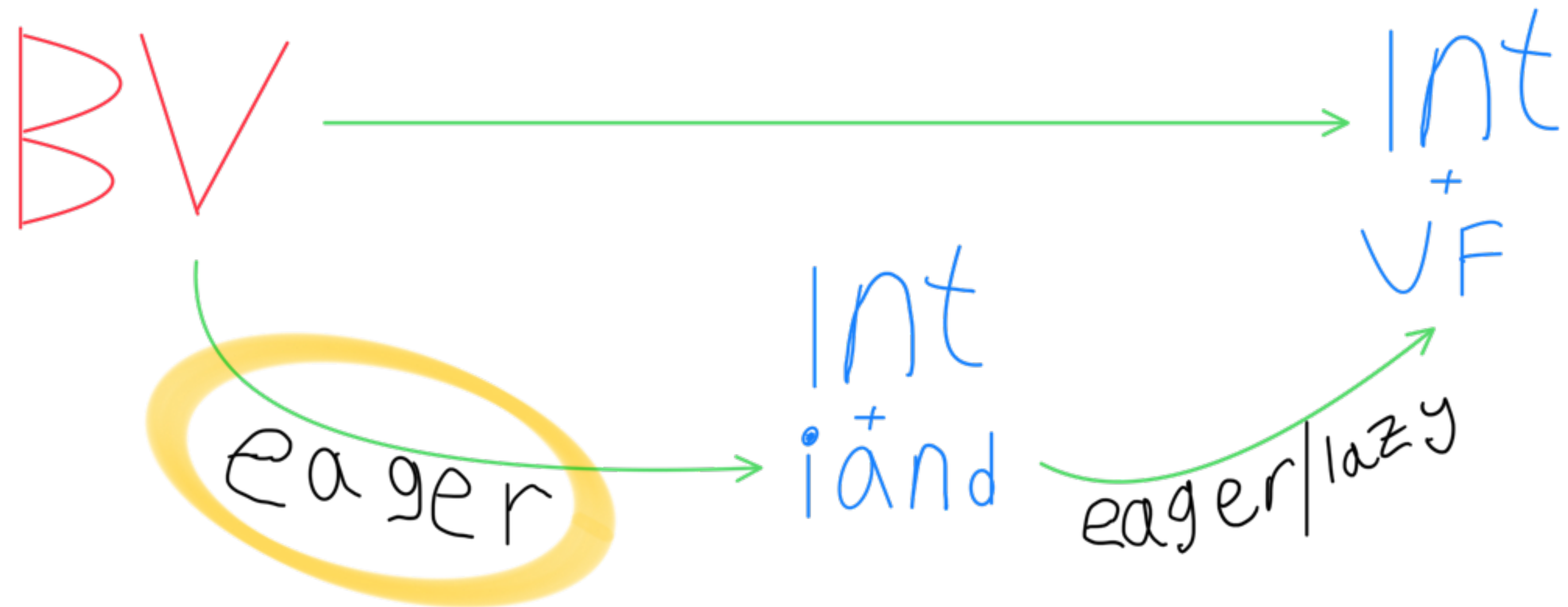
$\bowtie\, \in \{ <, \leq, >, \geq \}$

$uts_k(\cdot)$: from **unsigned** to **signed**

$$\begin{tikzcd} 9 \arrow[r] \arrow[rr, bend right, "uts_4"'] & 1001 \arrow[r] & -7 \end{tikzcd}$$

$$\mathsf{uts}_k(x) = 2 \cdot (x \bmod 2^{k-1}) - x$$

BV $\longrightarrow$ Arith + iand



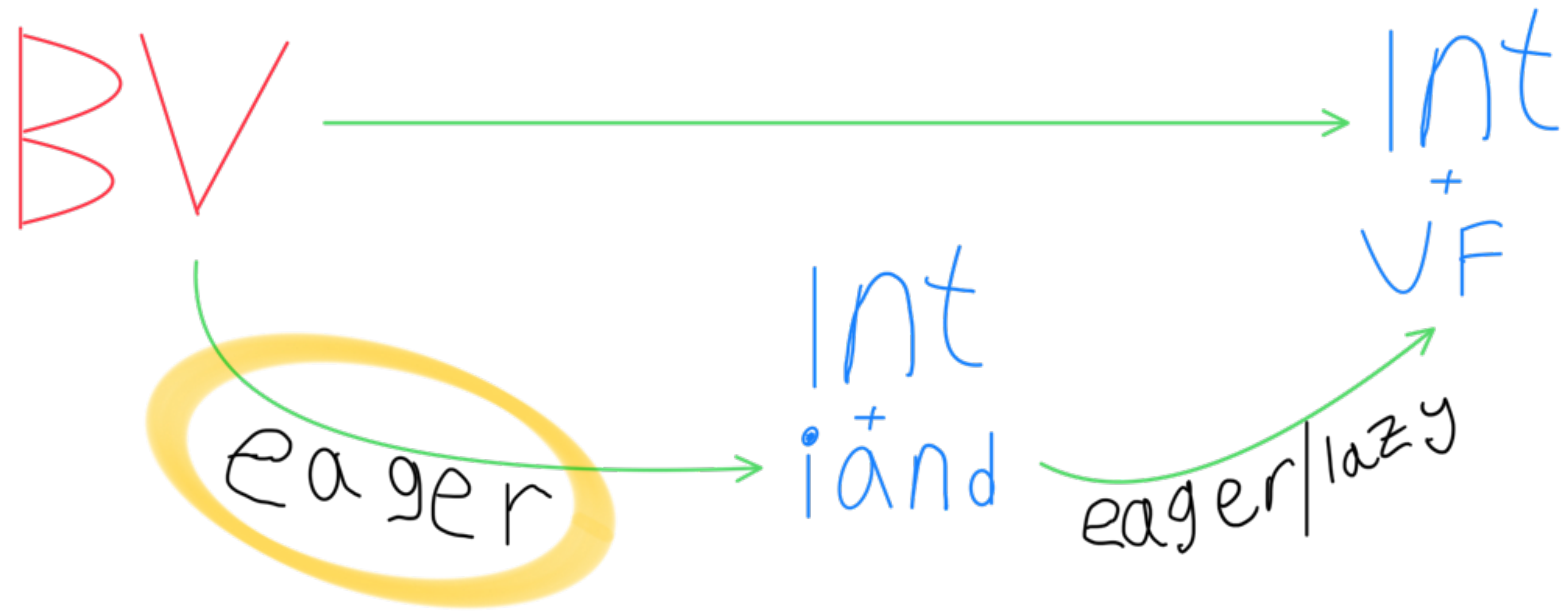
$\underline{\mathcal{T}\,\varphi}$:
$\mathcal{C}\,\varphi \wedge \textsc{Lem}^{\leq}(\varphi)$

$\underline{\mathcal{C}\,e}$:
Match e:

$$
\begin{array}{lcl}
t_1 \;+^{\mathrm{BV}} t_2 & \to & (\mathcal{C}\,t_1 + \mathcal{C}\,t_2) \bmod 2^k \\
t_1 \;-^{\mathrm{BV}} t_2 & \to & (\mathcal{C}\,t_1 - \mathcal{C}\,t_2) \bmod 2^k \\
t_1 \cdot^{\mathrm{BV}} t_2 & \to & (\mathcal{C}\,t_1 \cdot \mathcal{C}\,t_2) \bmod 2^k \\
\sim^{\mathrm{BV}} t_1 & \to & 2^k - (\mathcal{C}\,t_1 + 1) \\
-^{\mathrm{BV}} t_1 & \to & (2^k - \mathcal{C}\,t_1) \bmod 2^k
\end{array}
$$

k is the bit-width

BV ⟶ Arith + iand



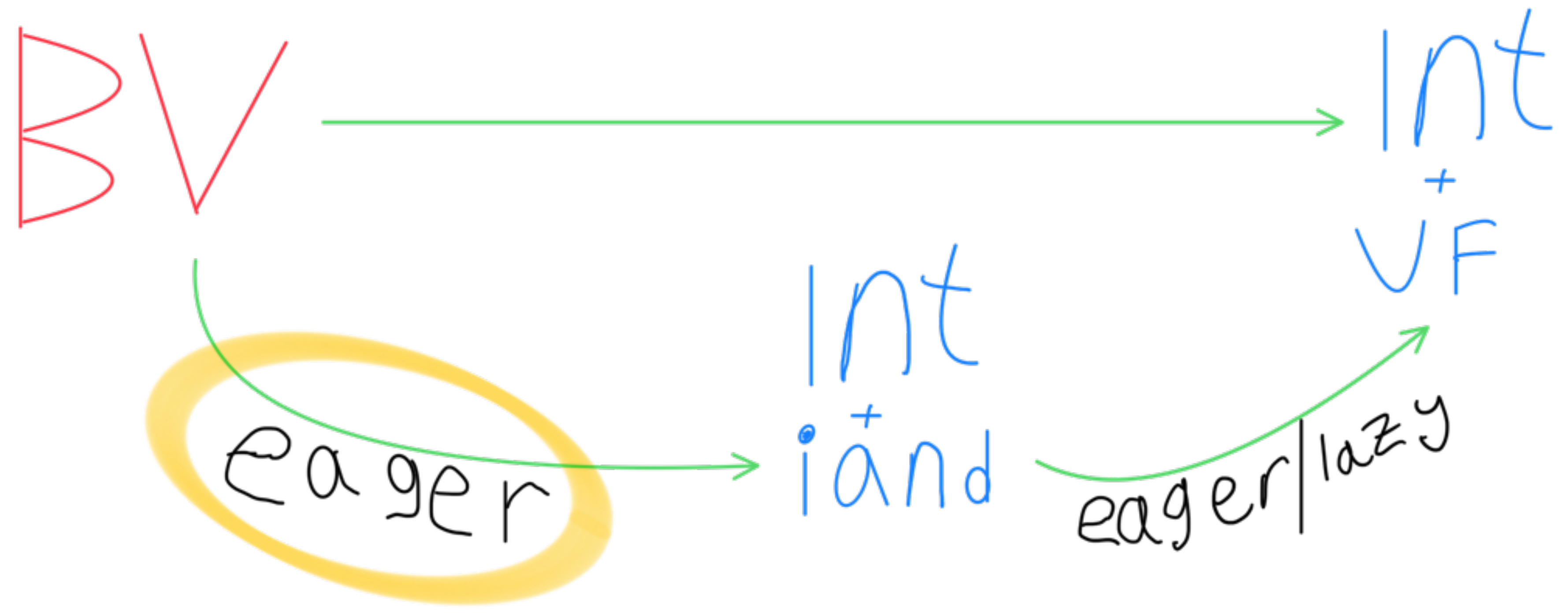
$\underline{\mathcal{T}\ \varphi}$:
$\mathcal{C}\ \varphi \wedge \textsc{Lem}^{\leq}(\varphi)$

$\underline{\mathcal{C}\ e}$:
Match e:

$$
\begin{array}{lcl}
t_1 \text{ div}^{\text{BV}} t_2 & \to & \text{ite}(\mathcal{C}\ t_2 = 0,\ 2^k - 1,\ \mathcal{C}\ t_1 \text{ div } \mathcal{C}\ t_2) \\
t_1 \text{ mod}^{\text{BV}} t_2 & \to & \text{ite}(\mathcal{C}\ t_2 = 0,\ \mathcal{C}\ t_1,\ \mathcal{C}\ t_1 \text{ mod } \mathcal{C}\ t_2) \\
t_1 \circ^{\text{BV}} t_2 & \to & \mathcal{C}\ t_1 \cdot 2^k + \mathcal{C}\ t_2 \\
t_1[u:l]^{\text{BV}} & \to & \mathcal{C}\ t_1 \text{ div } 2^l \text{ mod } 2^{u-l+1}
\end{array}
$$

ite — if then else

BV $\longrightarrow$ Arith + iand



$\underline{\mathcal{T}\ \varphi}$:

$\mathcal{C}\ \varphi \wedge \textsc{Lem}^{\leq}(\varphi)$

$\underline{\mathcal{C}\ e}$:

Match e:

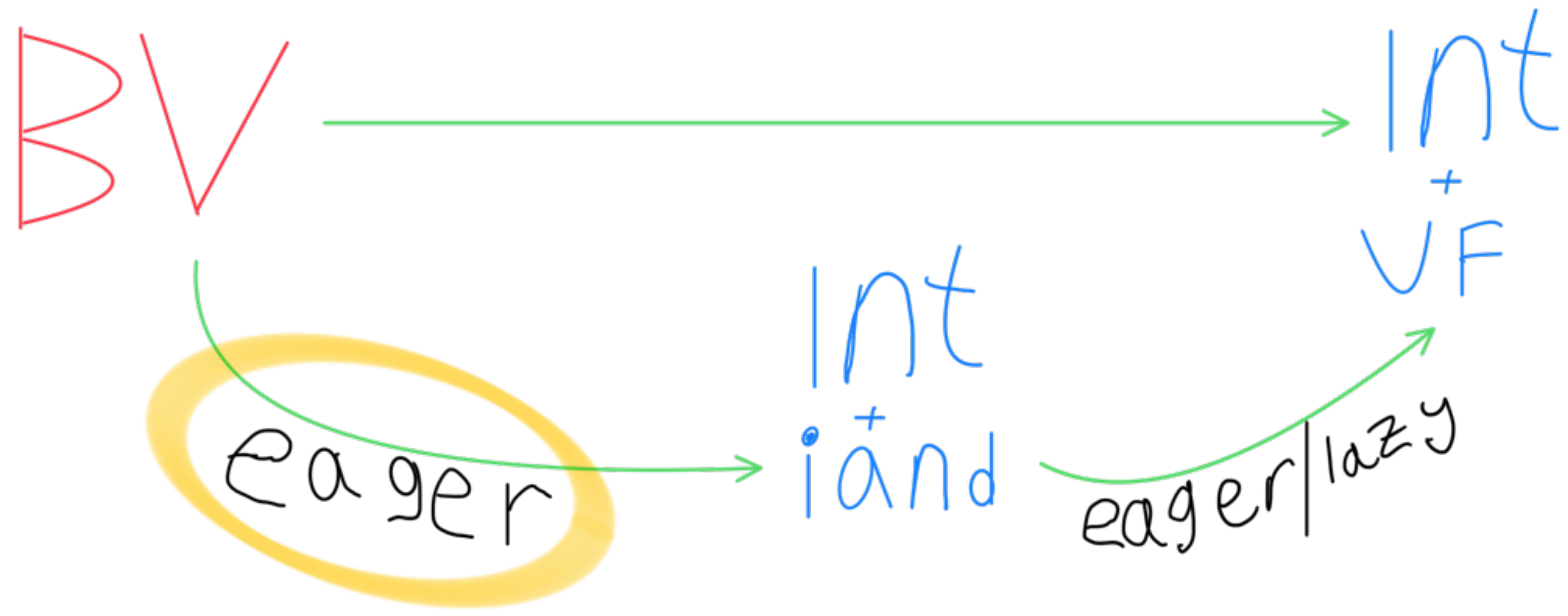
$$t_1 \ll^{\mathrm{BV}} t_2 \quad \to \quad (\mathcal{C}\ t_1 \cdot \mathrm{pow2}(\mathcal{C}\ t_2)) \bmod 2^k$$

$$t_1 \gg^{\mathrm{BV}} t_2 \quad \to \quad \mathcal{C}\ t_1 \ \mathrm{div}\ \mathrm{pow2}(\mathcal{C}\ t_2)$$

pow2 is eliminated using `ite`

$$pow2(x) = ite(x = 0, 1, ite(x = 1, 2, ite(\ldots, ite(x = k, 2^k, 0)\ldots)$$

$\mathbf{BV} \longrightarrow$ Arith + iand



$\underline{\mathcal{T}\,\varphi}$:

$\mathcal{C}\,\varphi \wedge \textsc{Lem}^{\leq}(\varphi)$

$\underline{\mathcal{C}\,e}$:

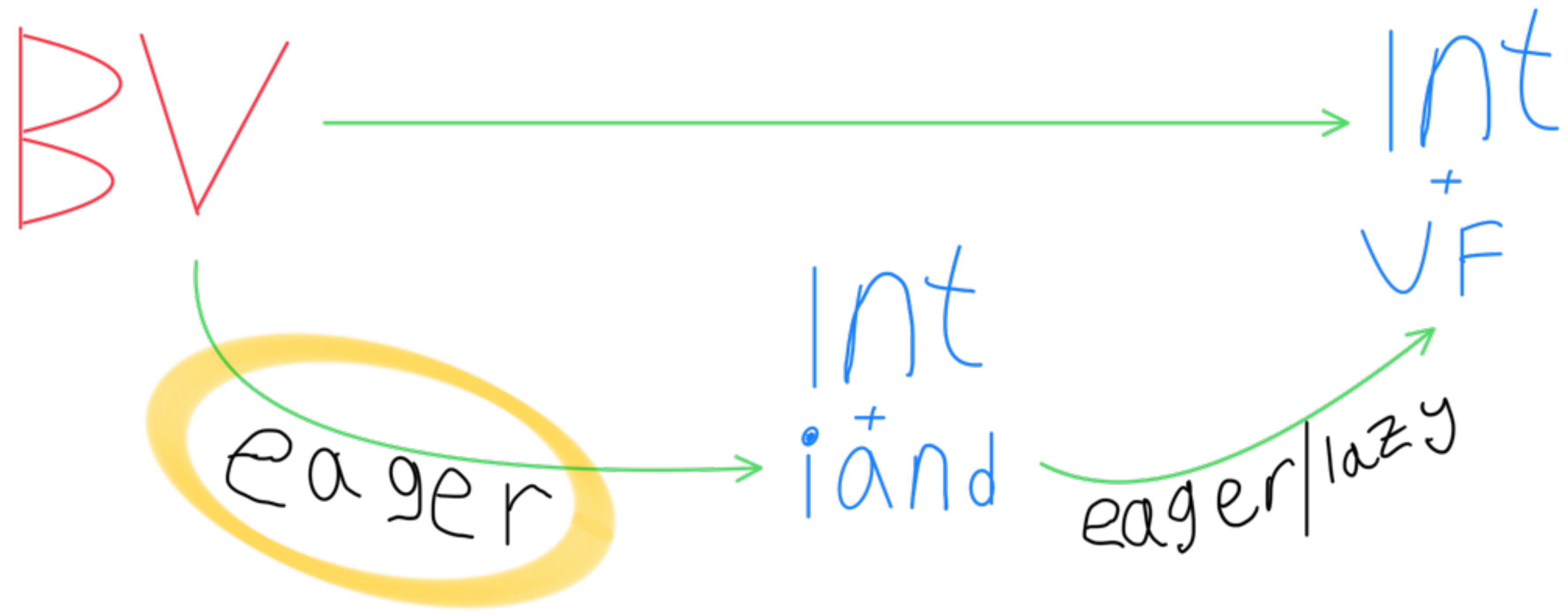
Match e:

$$t_1 \,\&^{\mathrm{BV}} t_2 \qquad \rightarrow \qquad \&_k^{\mathbb{N}}(\mathcal{C}\,t_1, \mathcal{C}\,t_2)$$

k is the bit-width

$\&_k^{\mathbb{N}}$ is an **iand** operator

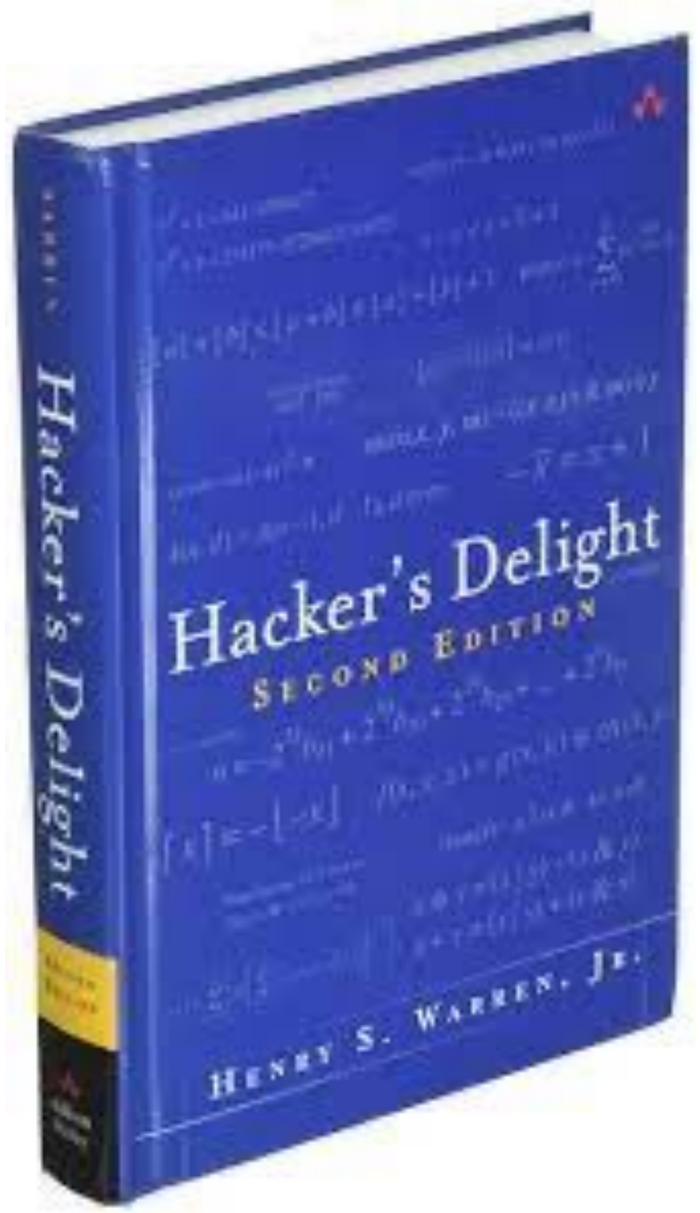
BV $\longrightarrow$ Arith + iand



$\underline{\mathcal{T}\,\varphi}$:
$\mathcal{C}\,\varphi \wedge \textsc{Lem}^{\leq}(\varphi)$

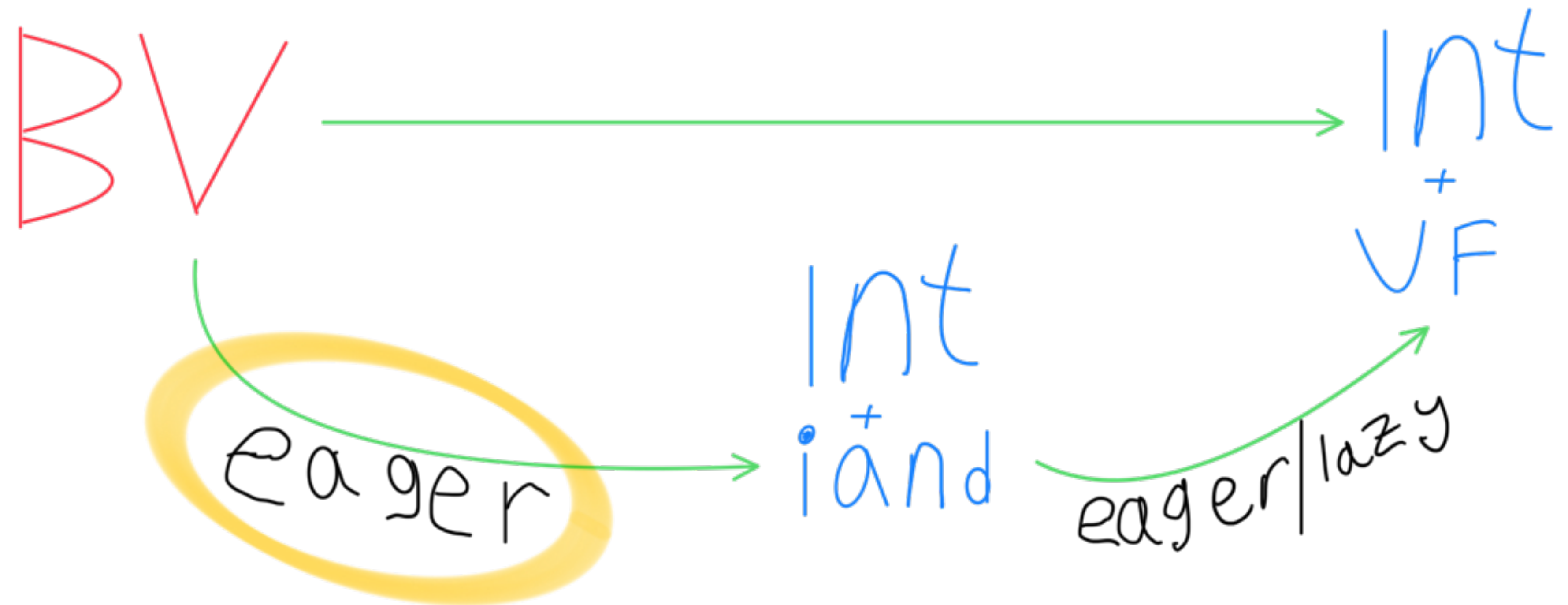
$\underline{\mathcal{C}\,e}$:
Match e:

$$x \mid^{\mathrm{BV}} y = (x +^{\mathrm{BV}} y) -^{\mathrm{BV}} (x \,\&^{\mathrm{BV}} y)$$
$$x \oplus^{\mathrm{BV}} y = (x \mid^{\mathrm{BV}} y) -^{\mathrm{BV}} (x \,\&^{\mathrm{BV}} y)$$



bvor and bvxor are eliminated

$\color{red}{\text{BV}}$ $\longrightarrow$ Arith + iand



$\mathcal{T}\,\varphi$:
$\mathcal{C}\,\varphi \wedge \textsc{Lem}^{\leq}(\varphi)$

$\textsc{Lem}^{\leq}(e)$:
Match e:

$$
\begin{array}{lll}
\boxed{x} & \rightarrow & 0 \leq \chi(x) < 2^{\kappa(x)} \\
c & \rightarrow & \top \\
t_1 = t_2 & \rightarrow & \textsc{Lem}^{\leq}(t_1) \wedge \textsc{Lem}^{\leq}(t_2) \\
f^{\mathrm{BV}}(t_1, t_2) & \rightarrow & 0 \leq \&^{\mathbb{N}}_{k}(\mathcal{C}\,t_1, \mathcal{C}\,t_2) < 2^{k} \wedge \\
 & & \textsc{Lem}^{\leq}(t_1) \wedge \textsc{Lem}^{\leq}(t_2) \\
g^{\mathrm{BV}}(t_1, \ldots, t_n) & \rightarrow & \bigwedge_{i=1}^{n} \textsc{Lem}^{\leq}(t_i) \\
\diamond(\varphi_1, \ldots, \varphi_n) & \rightarrow & \bigwedge_{i=1}^{n} \textsc{Lem}^{\leq}(\varphi_i)
\end{array}
$$

$LEM^{\leq}$ includes range constraints $0 \leq t < 2^k$

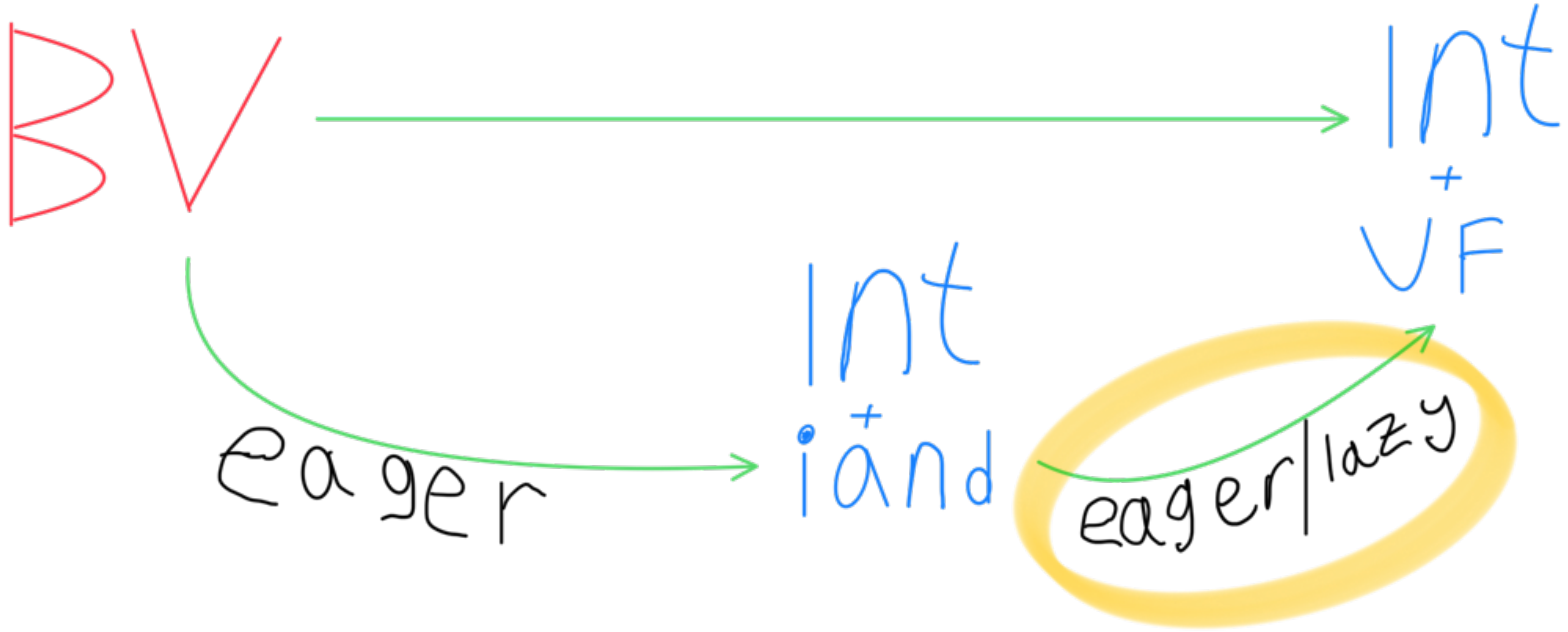
$f^{\mathbf{BV}} \in \{\, \&^{\mathbf{BV}}, |^{\mathbf{BV}}, \oplus^{\mathbf{BV}} \,\}$

$g^{\mathbf{BV}}$: other BV operators

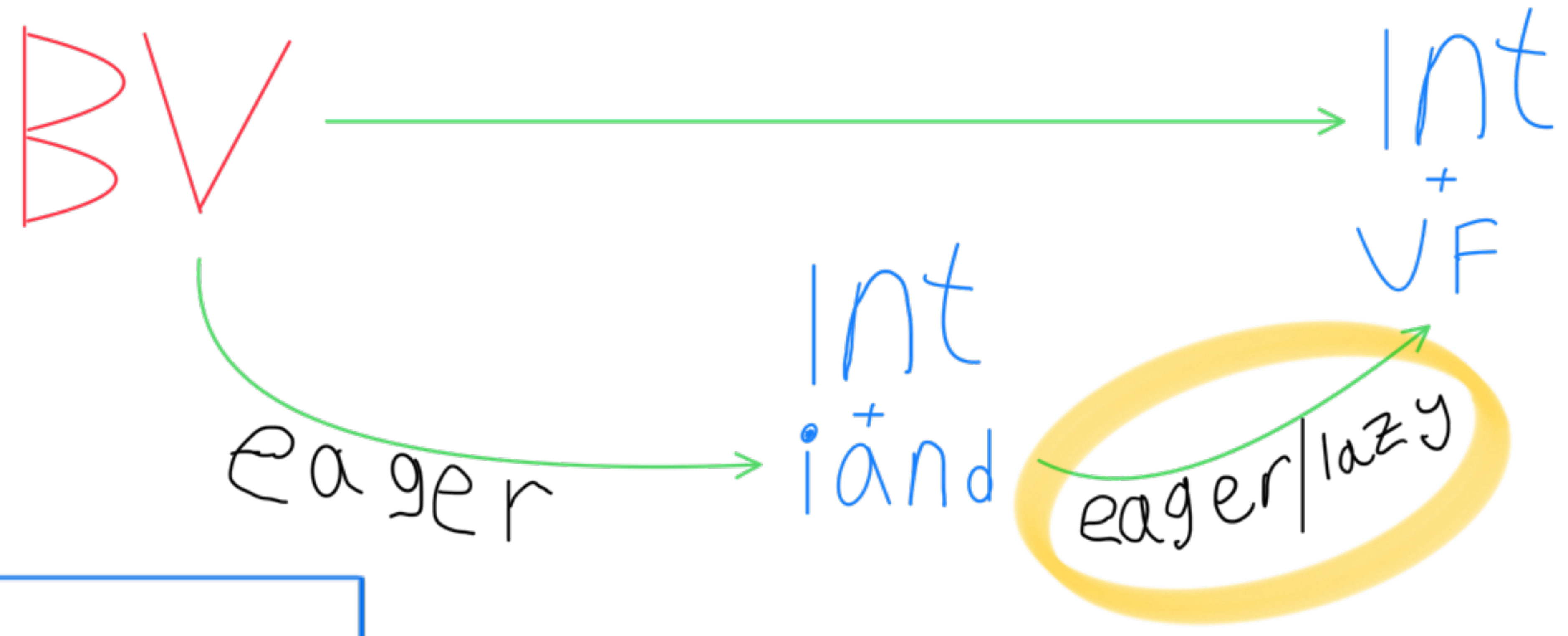
$\diamond$: Boolean operators

χ maps BV variables to integer variables

Int-blasting

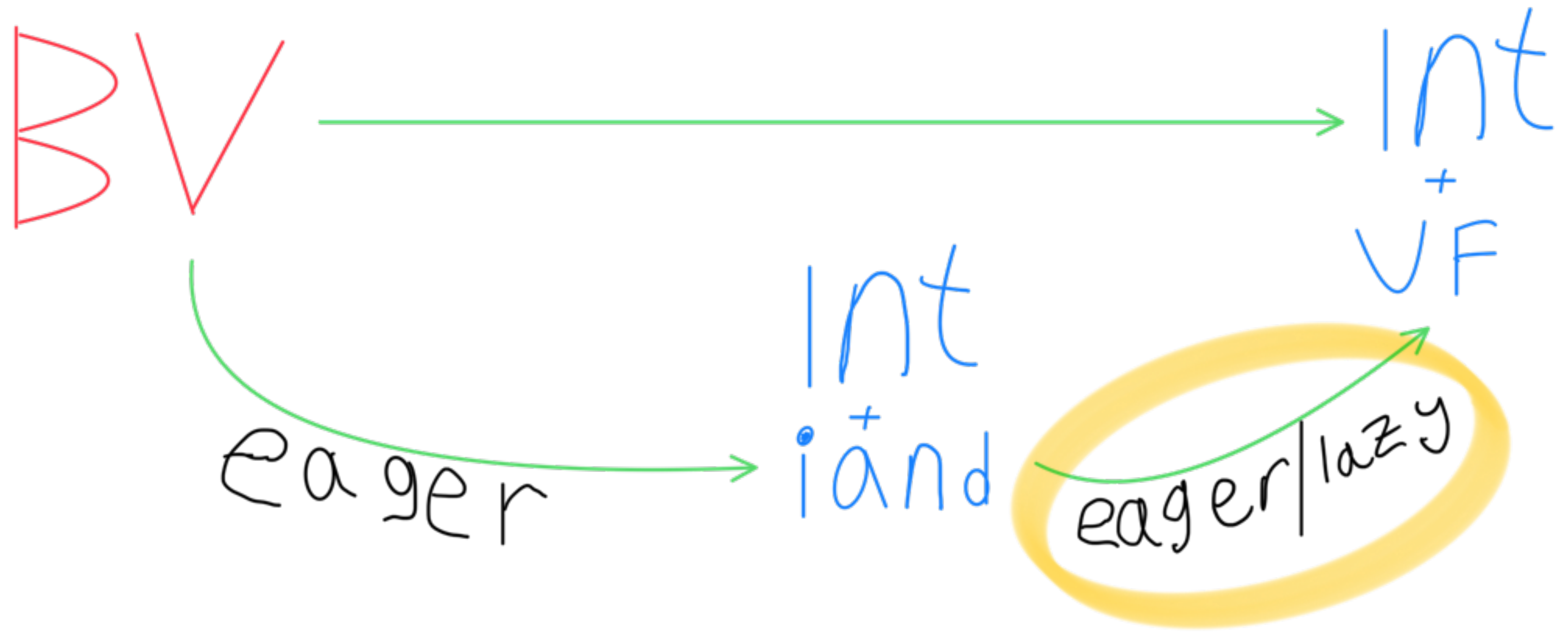


Arith + iand ⟶ Arith + UF



algorithm \ encoding	sum	bitwise
eager	$\bigwedge \sum(\ldots) = \ldots$	$\bigwedge \bigwedge \ldots = \ldots$
lazy	$\sum(\ldots) = \ldots$ ⋮ $\sum(\ldots) = \ldots$	$\bigwedge \ldots = \ldots$ ⋮ $\bigwedge \ldots = \ldots$

Arith + iand $\longrightarrow$ Arith + UF



Eager Version

$\underline{\mathcal{T}_A\,\varphi}$:

$\textsc{Lem}^{\&}_A(\varphi) \wedge \varphi$

$\underline{\textsc{Lem}^{\&}_A(e)}$:

Match e:

$$
\begin{array}{ll}
x & \to \top \\
c & \to \top \\
t_1 = t_2 & \to \textsc{Lem}^{\&}_A(t_1) \wedge \textsc{Lem}^{\&}_A(t_2) \\
\diamond(\varphi_1, \ldots, \varphi_n) & \to \bigwedge_{i=1}^{n} \textsc{Lem}^{\&}_A(\varphi_i) \\
f(t_1, \ldots, t_n) & \to \bigwedge_{i=1}^{n} \textsc{Lem}^{\&}_A(t_i) \\
\&^{\mathbb{N}}_k(t_1, t_2) & \to \boxed{\textsc{Iand}_A(t_1, t_2)} \wedge \bigwedge_{i \in \{1,2\}} \textsc{Lem}^{\&}_A(t_i)
\end{array}
$$

$A \in \{$**sum**, **bitwise**$\}$

Arith + iand $\longrightarrow$ Arith + UF

Eager-sum Version

$$
\begin{tikzcd}
\mathrm{BV} \arrow[rr] \arrow[d, bend right, "\text{eager}"'] & & \mathrm{Int} + \mathrm{UF} \\
& \mathrm{Int} + \mathrm{iand} \arrow[ur, "\text{eager}|\text{lazy}"'] &
\end{tikzcd}
$$

$\underline{\mathcal{T}_A\ \varphi}$:

$\textsc{Lem}_A^{\&}(\varphi) \wedge \varphi$

$\underline{\textsc{Lem}_A^{\&}(e)}$:

Match e:

$$
\begin{array}{ll}
x & \rightarrow \top \\
c & \rightarrow \top \\
t_1 = t_2 & \rightarrow \textsc{Lem}_A^{\&}(t_1) \wedge \textsc{Lem}_A^{\&}(t_2) \\
\diamond(\varphi_1, \ldots, \varphi_n) & \rightarrow \bigwedge_{i=1}^{n} \textsc{Lem}_A^{\&}(\varphi_i) \\
f(t_1, \ldots, t_n) & \rightarrow \bigwedge_{i=1}^{n} \textsc{Lem}_A^{\&}(t_i) \\
\&_k^{\mathbb{N}}(t_1, t_2) & \rightarrow \boxed{\textsc{Iand}_A(t_1, t_2)} \wedge \bigwedge_{i \in \{1,2\}} \textsc{Lem}_A^{\&}(t_i)
\end{array}
$$

$A \in \{\textbf{sum}, \textbf{bitwise}\}$

$\underline{\textsc{Iand}_{\textsf{sum}}(t_1, t_2)}$:

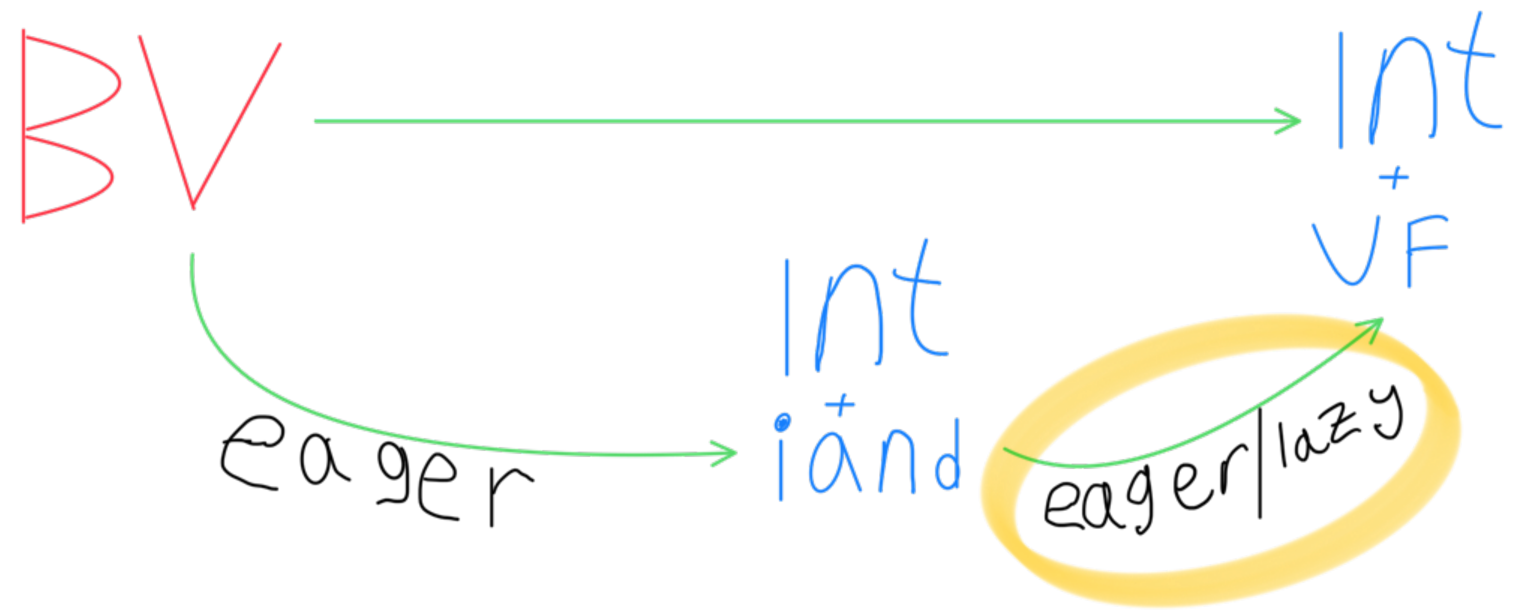
$$\&_k^{\mathbb{N}}(t_1, t_2) = \Sigma_{i=0}^{k-1} 2^i \cdot \mathrm{ITE}(a_i, b_i)$$

$$a_i = t_1\ div\ 2^i\ mod\ 2$$
$$b_i = t_2\ div\ 2^i\ mod\ 2$$
$$ITE(x, y) = ite(x = y = 1, 1, 0)$$

a_i **: ith bit of** t_1

b_i **: ith bit of** t_2

Arith + iand $\longrightarrow$ Arith + UF



Eager-bitwise Version

$\underline{\mathcal{T}_A\ \varphi}$:

$\textsc{Lem}^{\&}_{A}(\varphi) \wedge \varphi$

$\underline{\textsc{Lem}^{\&}_{A}(e)}$:

Match e:

$$\begin{array}{ll}
x & \to \top \\
c & \to \top \\
t_1 = t_2 & \to \textsc{Lem}^{\&}_{A}(t_1) \wedge \textsc{Lem}^{\&}_{A}(t_2) \\
\diamond(\varphi_1, \ldots, \varphi_n) & \to \bigwedge_{i=1}^{n} \textsc{Lem}^{\&}_{A}(\varphi_i) \\
f(t_1, \ldots, t_n) & \to \bigwedge_{i=1}^{n} \textsc{Lem}^{\&}_{A}(t_i) \\
\&^{\mathbb{N}}_{k}(t_1, t_2) & \to \boxed{\textsc{Iand}_A(t_1, t_2)} \wedge \bigwedge_{i \in \{1,2\}} \textsc{Lem}^{\&}_{A}(t_i)
\end{array}$$

$A \in \{$**sum**, **bitwise**$\}$

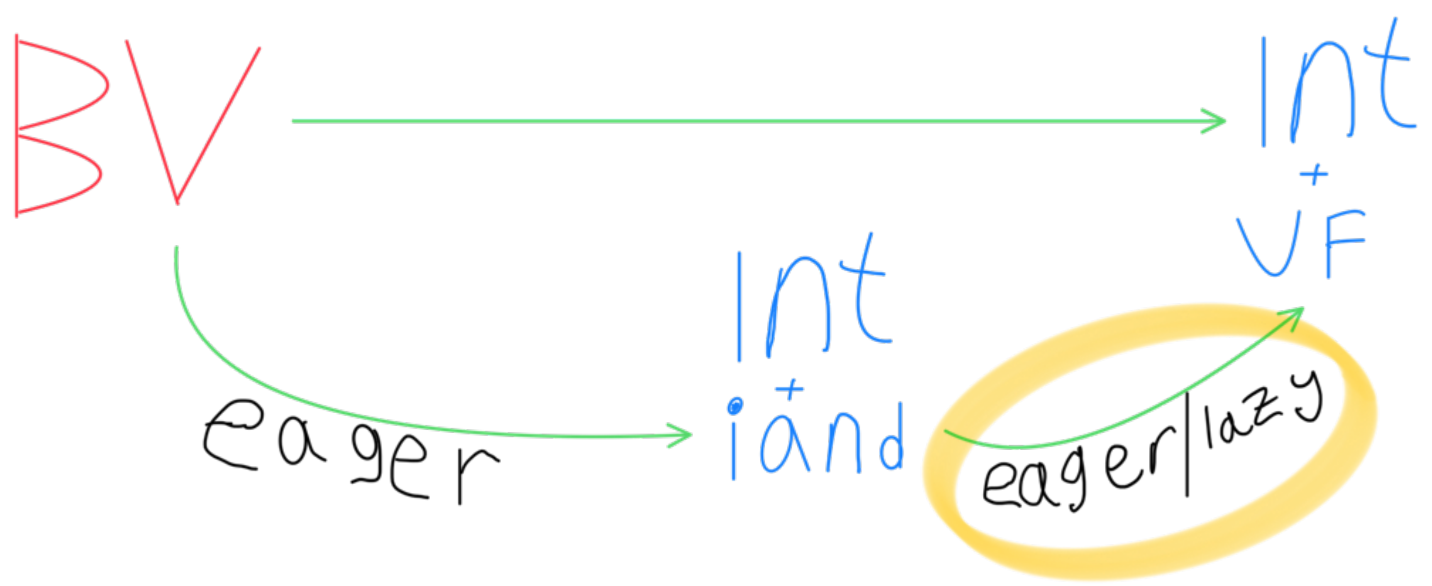
$\underline{\textsc{Iand}_{\text{bitwise}}(t_1, t_2)}$:

$$\bigwedge_{i=0}^{k-1} c_i = \mathrm{ITE}(a_i, b_i)$$

$$a_i = t_1\ div\ 2^i\ mod\ 2$$
$$b_i = t_2\ div\ 2^i\ mod\ 2$$
$$c_i = \&^{\mathbb{N}}_{k}(t_1, t_2)\ div\ 2^i\ mod\ 2$$

$$ITE(x, y) = ite(x = y = 1, 1, 0)$$

Arith + iand $\longrightarrow$ Arith + UF



Lazy Versions

$\Gamma := \{\, \mathcal{T}\,\varphi \,\}$

$\Delta := \{\, \&_k^{\mathbb{N}}(t_1, t_2) \mid \&_k^{\mathbb{N}}(t_1, t_2) \text{ occurs in } \mathcal{T}\,\varphi \,\}$

$\Lambda := \boxed{Prop(\Delta)} \cup \{\, \boxed{\mathrm{IAND}_{\mathcal{A}}(t_1, t_2)} \mid \&_k^{\mathbb{N}}(t_1, t_2) \in \Delta \,\}$

Repeat:

1. If $P_{T_{\mathrm{IAUF}}}(\bigwedge \Gamma)$ is "unsat", then return "unsat".
2. Otherwise, let $\mathcal{I} = P_{T_{\mathrm{IAUF}}}(\bigwedge \Gamma)$

   /* check $\mathcal{I}$ against properties of $\&_k^{\mathbb{N}}$ */

   (a) If $\mathcal{I}$ satisfies Λ, return "sat".

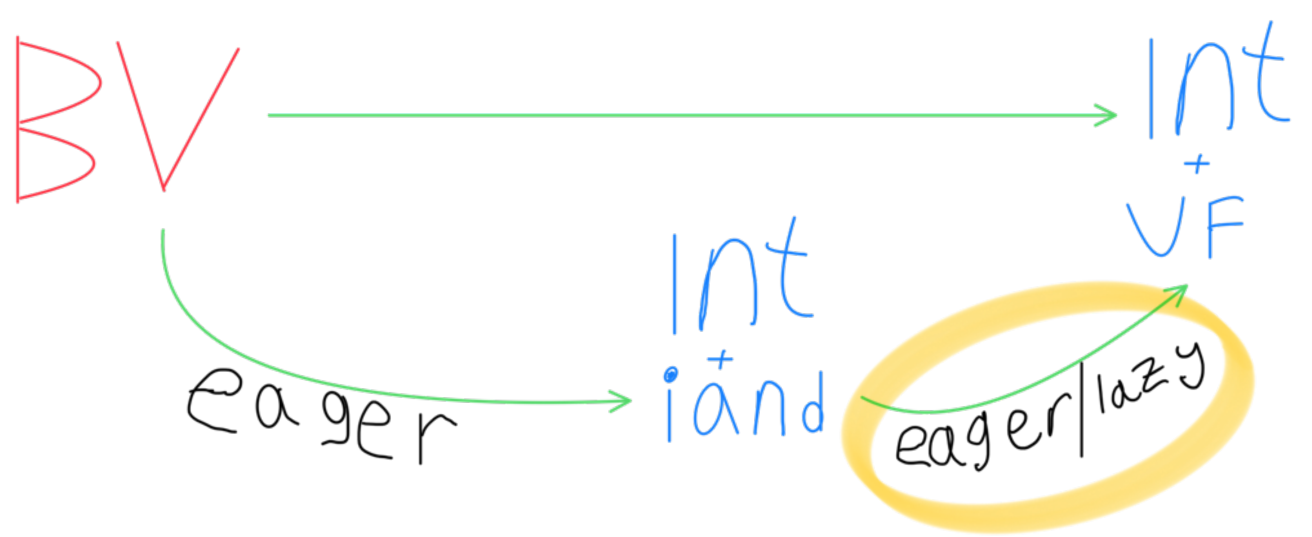
   (b) Otherwise:

   /* refine abstraction Γ */

   $\Gamma := \Gamma \cup \{\, \psi \in \Lambda \mid \mathcal{I} \not\models \psi \,\}$

$P_{T_{IAUF}}$ **is a UFNIA solver**

Arith + iand $\longrightarrow$ Arith + UF



Lazy Versions

$\Gamma := \{\, \mathcal{T}\,\varphi \,\}$

$\Delta := \{\, \&_k^{\mathbb{N}}(t_1, t_2) \mid \&_k^{\mathbb{N}}(t_1, t_2) \text{ occurs in } \mathcal{T}\,\varphi \,\}$

$\Lambda := \boxed{Prop(\Delta)} \cup \{\, \boxed{\text{IAND}_A(t_1, t_2)} \mid \&_k^{\mathbb{N}}(t_1, t_2) \in \Delta \,\}$

Repeat:

1. If $P_{T_{\text{IAUF}}}(\bigwedge \Gamma)$ is "unsat", then return "unsat".
2. Otherwise, let $\mathcal{I} = P_{T_{\text{IAUF}}}(\bigwedge \Gamma)$

   /* check $\mathcal{I}$ against properties of $\&_k^{\mathbb{N}}$ */

   (a) If $\mathcal{I}$ satisfies Λ, return "sat".

   (b) Otherwise:

   /* refine abstraction Γ */

   $\Gamma := \Gamma \cup \{\, \psi \in \Lambda \mid \mathcal{I} \not\models \psi \,\}$

$Prop(\Delta) = \{\, Prop(t_1, t_2) \mid \&_k^{\mathbb{N}}(t_1, t_2) \in \Delta \,\}$

$Prop(t_1, t_2)$:

$\&_k^{\mathbb{N}}(t_1, t_2) \le t_1 \;\wedge\; \&_k^{\mathbb{N}}(t_1, t_2) \le t_2 \;\wedge$	bounds
$(t_1 = t_2 \Rightarrow \&_k^{\mathbb{N}}(t_1, t_2) = t_1) \;\wedge$	idempotence
$\&_k^{\mathbb{N}}(t_1, t_2) = \&_k^{\mathbb{N}}(t_2, t_1) \;\wedge$	symmetry
$\left.\begin{array}{l}(t_1 = 0 \Rightarrow \&_k^{\mathbb{N}}(t_1, t_2) = 0) \;\wedge \\ (t_1 = 2^k - 1 \Rightarrow \&_k^{\mathbb{N}}(t_1, t_2) = t_2) \;\wedge \\ (t_2 = 0 \Rightarrow \&_k^{\mathbb{N}}(t_1, t_2) = 0) \;\wedge \\ (t_2 = 2^k - 1 \Rightarrow \&_k^{\mathbb{N}}(t_1, t_2) = t_1)\end{array}\right\}$	special cases

$\underline{\text{IAND}_{\textsf{sum}}(t_1, t_2)}$:

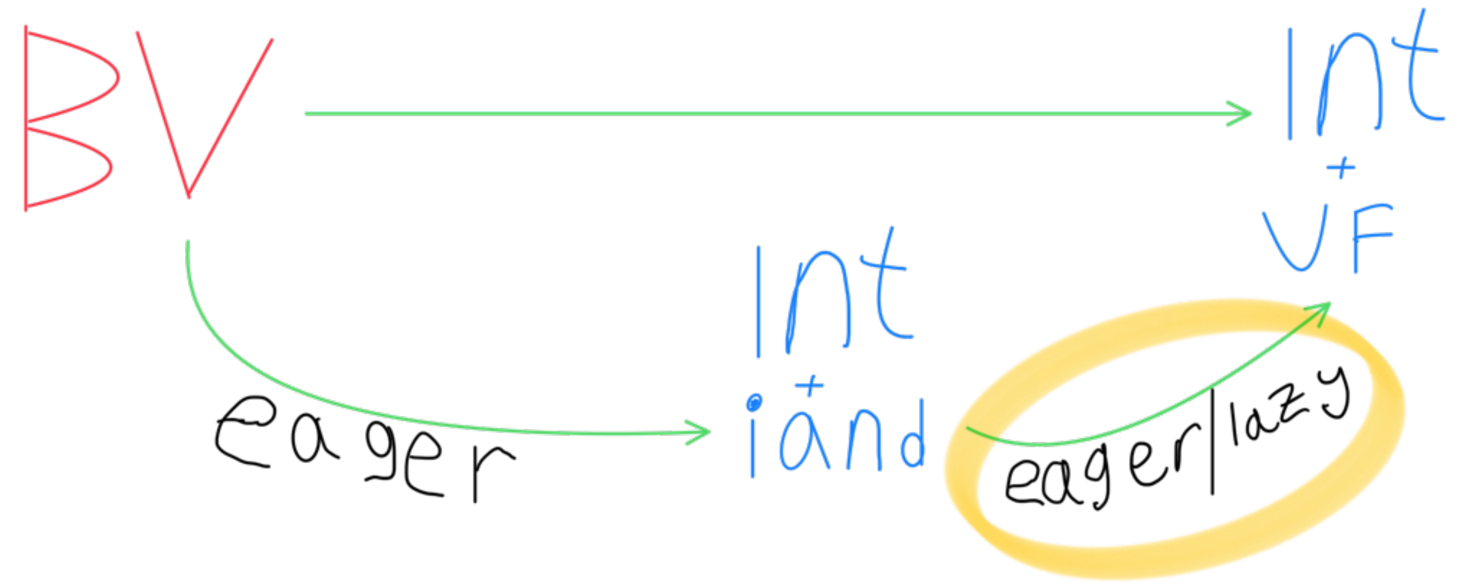
$$\&_k^{\mathbb{N}}(t_1, t_2) = \Sigma_{i=0}^{k-1} 2^i \cdot \text{ITE}(a_i, b_i)$$

$\underline{\text{IAND}_{\textsf{bitwise}}(t_1, t_2)}$:

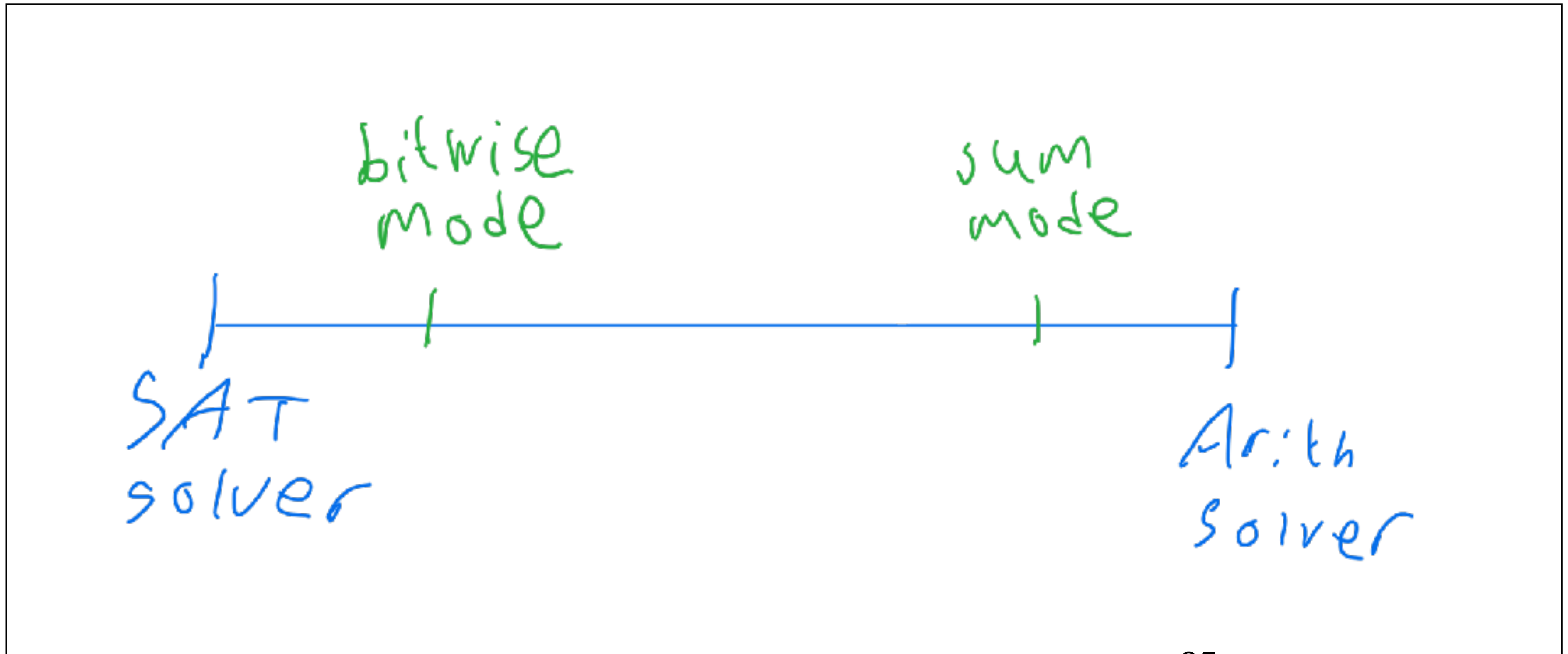
$$\bigwedge_{i=0}^{k-1} c_i = \text{ITE}(a_i, b_i)$$

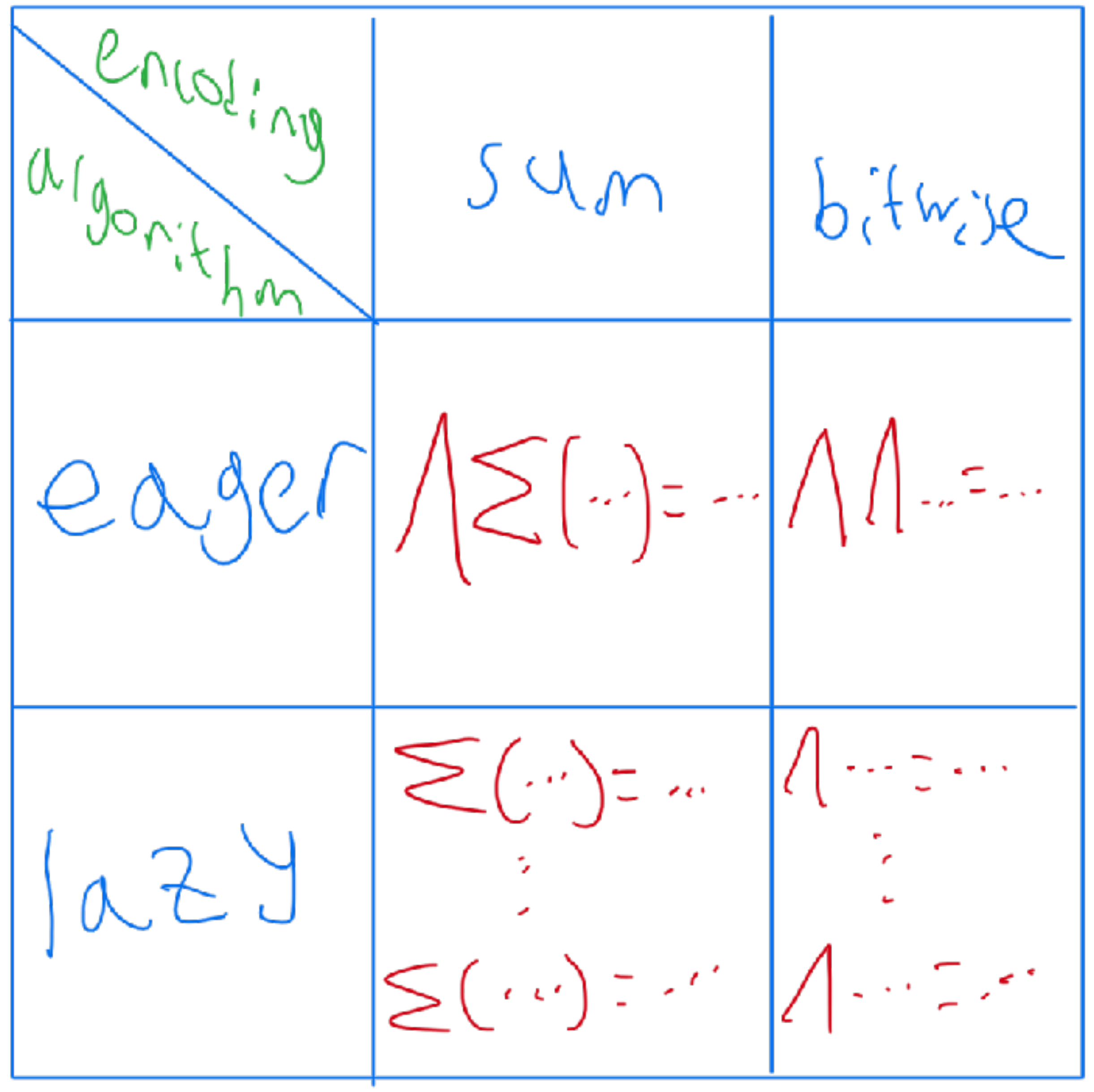
$P_{T_{IAUF}}$ **is a UFNIA solver**

Arith + iand $\longrightarrow$ Arith + UF



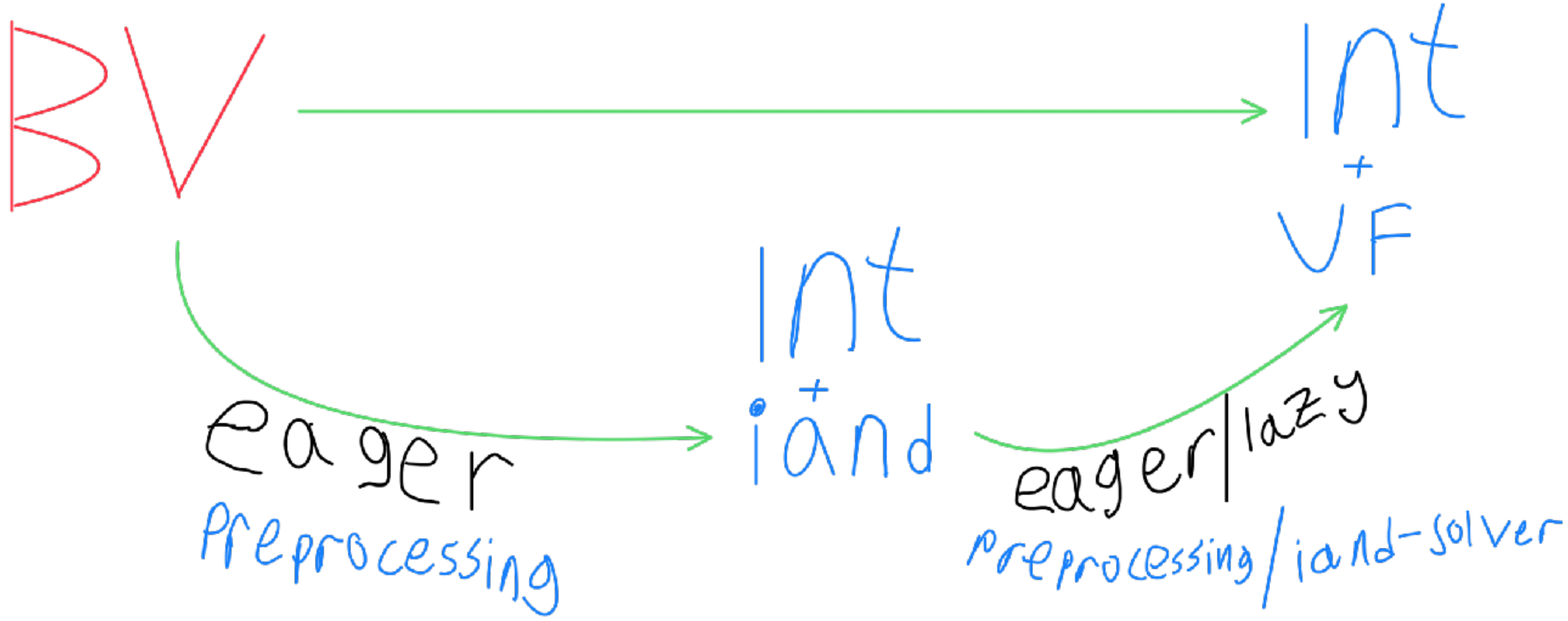
- Both modes utilize the SAT-solver and Arith-solver
    - “The ith bit of x” — $(x\ div\ 2^i)\ mod\ 2$
- *bitwise* mode relies more on the SAT-solver
- *sum* mode relies more on the Arith-solver





Evaluation

Int-blasting is implemented in cvc5 (successor of CVC4)



Evaluation

- Other tools:
  - Bitwuzla — first place in QF_BV 2020
  - Yices — second place in QF_BV 2020
  - cvc5 eager bit-blaster — baseline
  - (bw-ind — our integer-based bit-width independent prototype)
- Benchmarks:
  - SMT-LIB
  - Rewrite-rule Candidates
  - Certora Smart Contracts Verification

SMT-LIB

- QF_BV family
- 41,713 benchmarks
- Very diverse
- Not many large bit-widths

SMT-LIB
THE SATISFIABILITY MODULO THEORIES LIBRARY

Rewrite Rule Candidates

- Hand-crafted but represents a real application — rewrite rules for SMT-solvers
- Benchmark generation using SyGuS:
  - Synthesize pairs of terms that are equivalent for bit-width 4
  - Prove correctness for larger bit-widths

- Benchmarks:
  - **5491** equivalence checks
  - Each one instantiated with **10** bit-widths (16, 32, … 8192)
  - Total **54,910** benchmarks

Smart Contracts Verification

- 35 benchmarks
- Given to us by Certora team
- QF_UFBV benchmarks with 256-bit bit-vectors
- Employ arithmetic and bitwise operators
- Encode algebraic properties (e.g., commutativity) of low-level methods

Results

	SMT-LIB				ECRW				SC			
	slvd	*sat*	*uns*	*m*	*slvd*	*sat*	*uns*	*m*	*slvd*	*sat*	*uns*	*m*
$eager_b$	35031	10447	24584	38	41989	119	41870	0	**24**	9	15	0
$eager_s$	35035	10459	24576	28	41435	119	41316	77	**24**	9	15	0
$lazy_b$	35001	10383	24618	23	**47071**	119	46952	0	**24**	9	15	0
$lazy_s$	34819	10297	24522	27	45350	119	45231	138	**24**	9	15	0
Bitwuzla	41220	14233	26987	19	37297	265	37032	11120	16	8	8	0
cvc5	40543	14204	26339	36	33187	220	32967	17535	-	-	-	-
Yices	**41228**	14280	26948	11	31646	255	31391	15801	9	3	6	0
bw-ind	-	-	-	-	25608	0	25608	0	-	-	-	-

Results — SMTLIB

- Timeout: 10 minutes
- Not competative on SMT-LIB
  - Expected — Bit-blasting is state of the art
- Better on UNSAT than on SAT
  - Expected — Lemmas are aimed at finding conflicts

	SMT-LIB				ECRW				SC			
	slvd	*sat*	*uns*	*m*	*slvd*	*sat*	*uns*	*m*	*slvd*	*sat*	*uns*	*m*
$eager_b$	35031	10447	24584	38	41989	119	41870	0	**24**	9	15	0
$eager_s$	35035	10459	24576	28	41435	119	41316	77	**24**	9	15	0
$lazy_b$	35001	10383	24618	23	**47071**	119	46952	0	**24**	9	15	0
$lazy_s$	34819	10297	24522	27	45350	119	45231	138	**24**	9	15	0
Bitwuzla	41220	14233	26987	19	37297	265	37032	11120	16	8	8	0
cvc5	40543	14204	26339	36	33187	220	32967	17535	-	-	-	-
Yices	**41228**	14280	26948	11	31646	255	31391	15801	9	3	6	0
bw-ind	-	-	-	-	25608	0	25608	0	-	-	-	-

SMT-LIB
THE SATISFIABILITY MODULO THEORIES LIBRARY

Results — Rewrite Rules

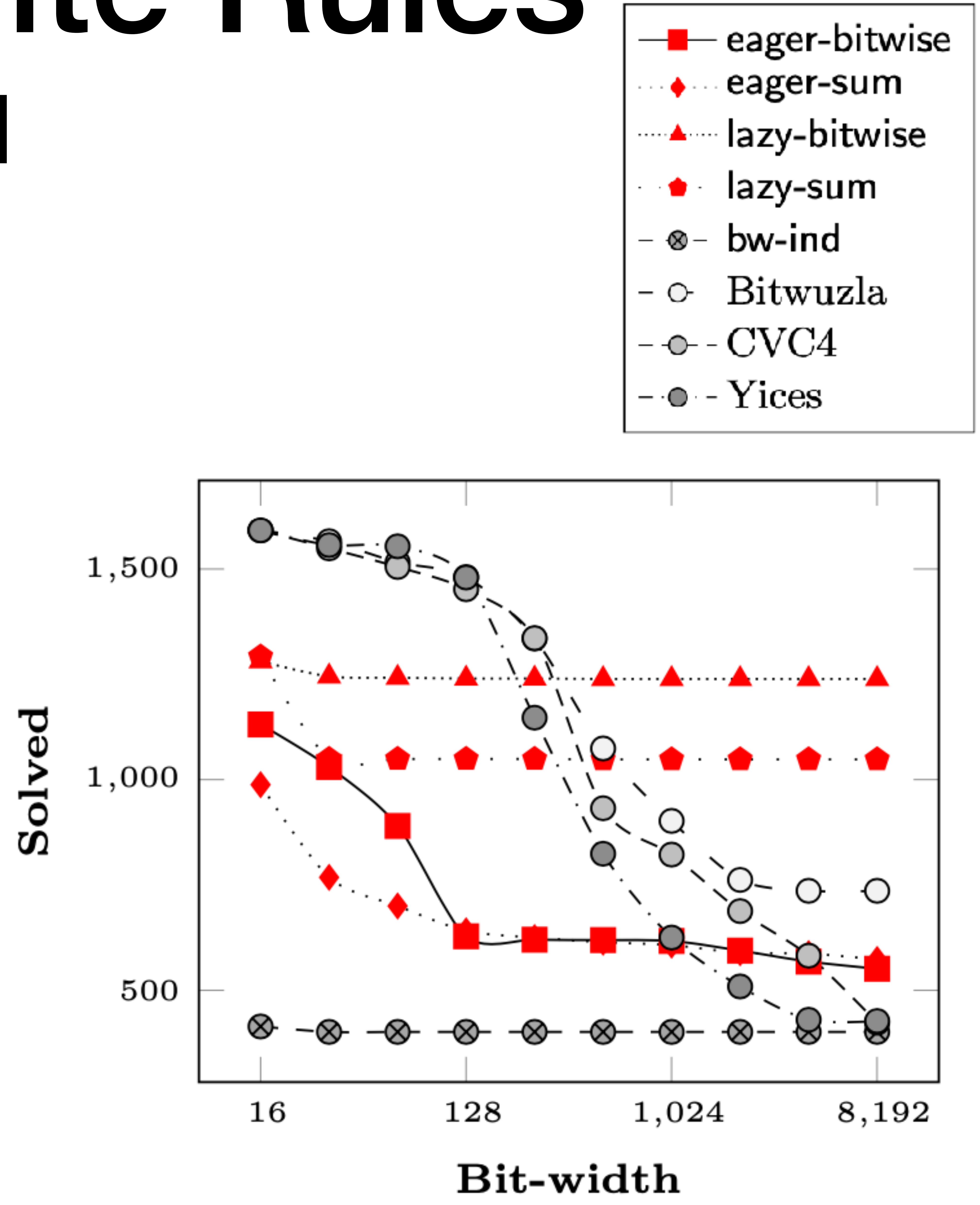
- Timeout: 5 minutes
- All int-blasting approaches are better
- Best int-blasting approach: lazy bitwise

	SMT-LIB				ECRW				SC			
	slvd	*sat*	*uns*	*m*	*slvd*	*sat*	*uns*	*m*	*slvd*	*sat*	*uns*	*m*
$eager_b$	35031	10447	24584	38	41989	119	41870	0	**24**	9	15	0
$eager_s$	35035	10459	24576	28	41435	119	41316	77	**24**	9	15	0
$lazy_b$	35001	10383	24618	23	**47071**	119	46952	0	**24**	9	15	0
$lazy_s$	34819	10297	24522	27	45350	119	45231	138	**24**	9	15	0
Bitwuzla	41220	14233	26987	19	37297	265	37032	11120	16	8	8	0
cvc5	40543	14204	26339	36	33187	220	32967	17535	-	-	-	-
Yices	**41228**	14280	26948	11	31646	255	31391	15801	9	3	6	0
bw-ind	-	-	-	-	25608	0	25608	0	-	-	-	-

Results — Rewrite Rules

With bvand

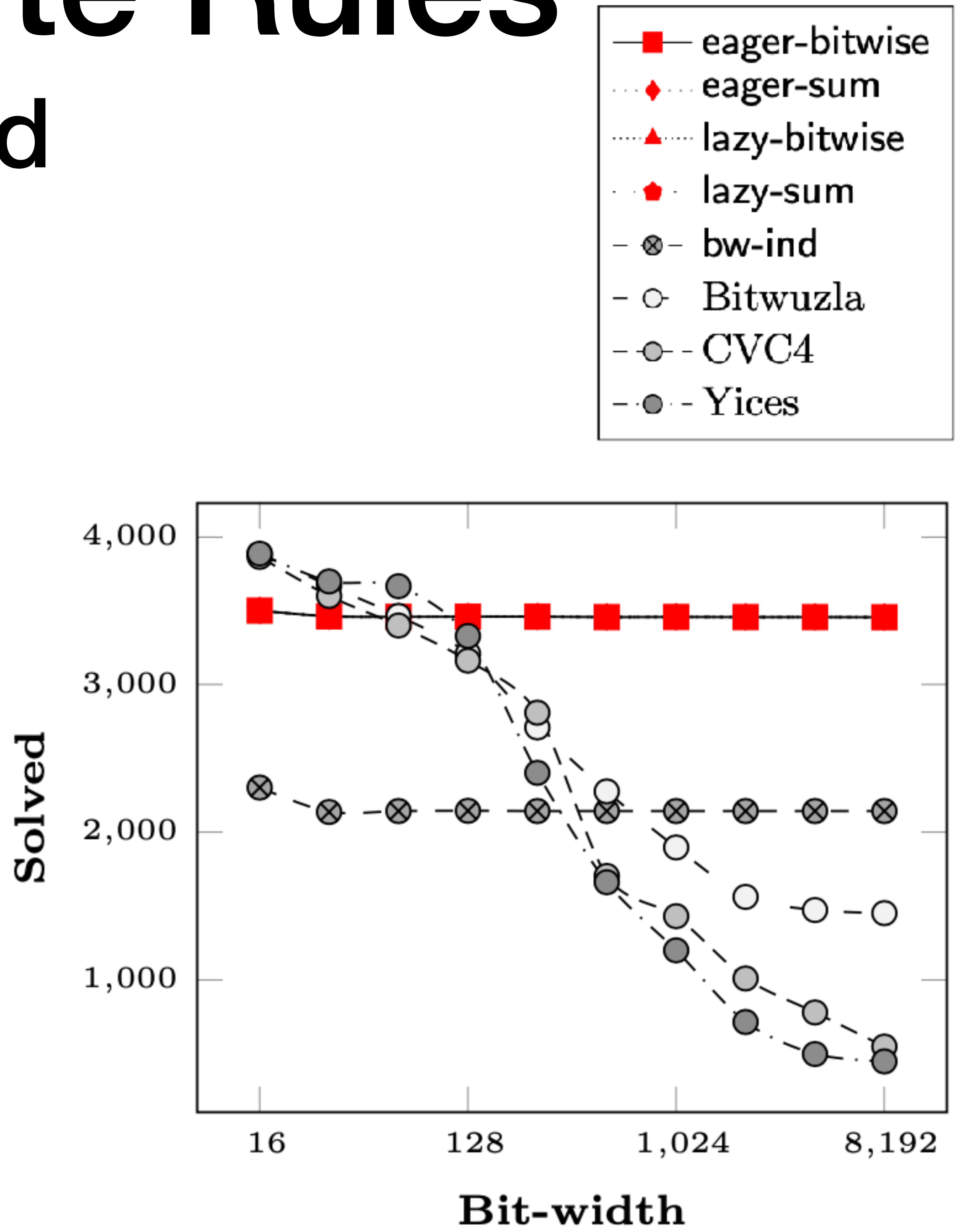
- with bvand: best starting from bit-width 512
- int-blasting approaches differ
- Lazy approaches are bit-width independent



Results — Rewrite Rules

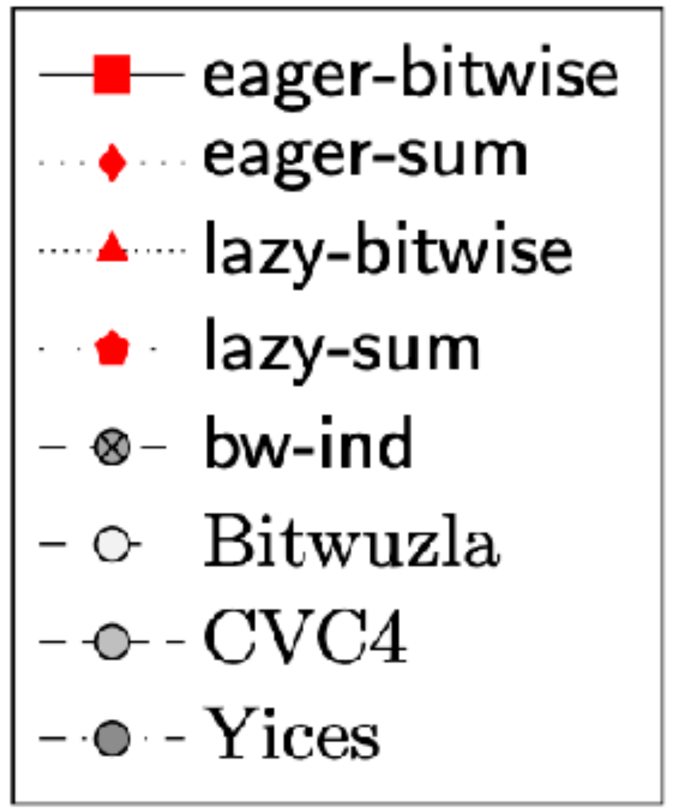
Without bvand

- with bvand: best starting from bit-width 128
- int-blasting approaches are identical
- bit-width independent

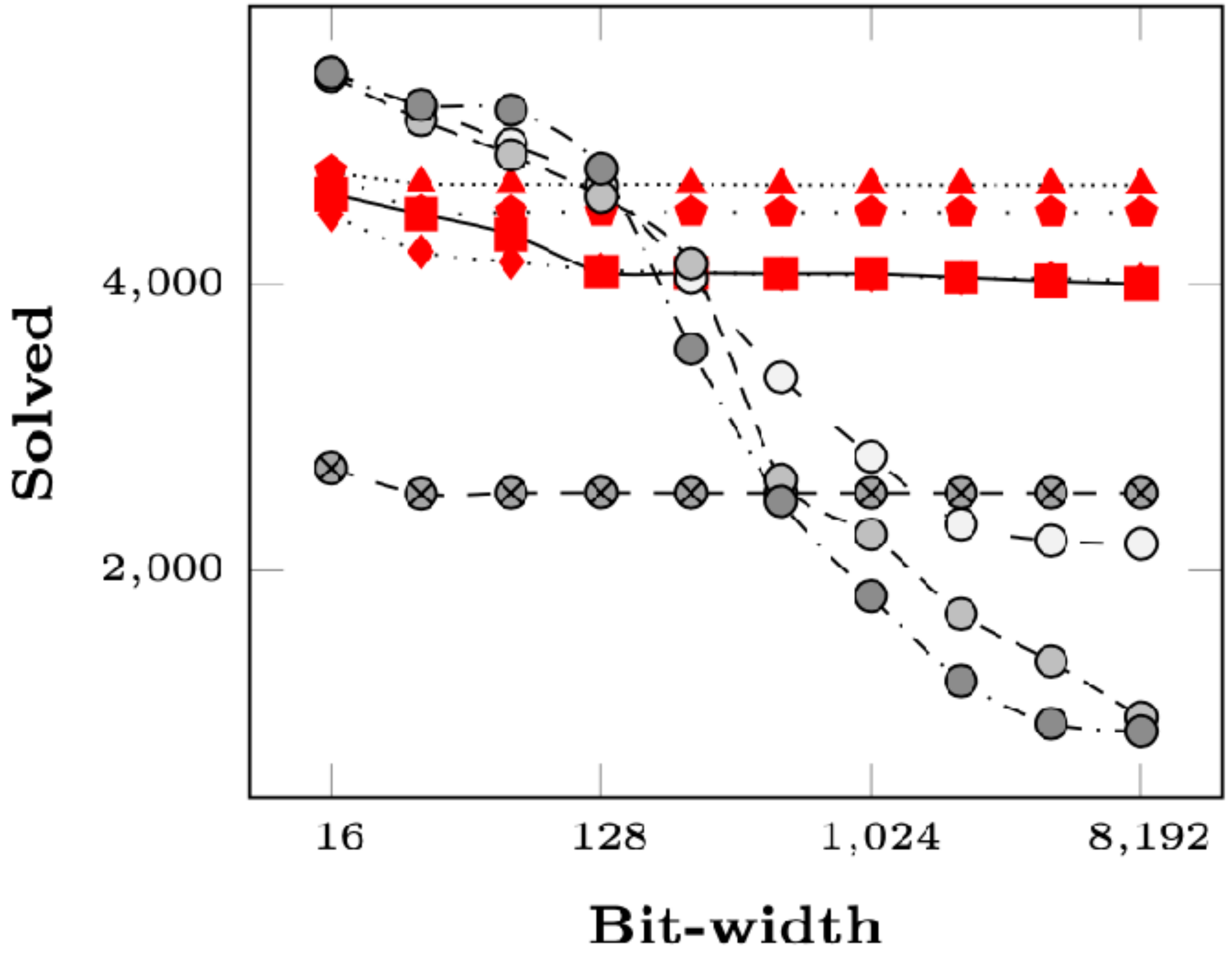


Results — Rewrite Rules

Full Set



- Full set: best starting from bit-width 256
- int-blasting approaches are similar
- Almost bit-width independent



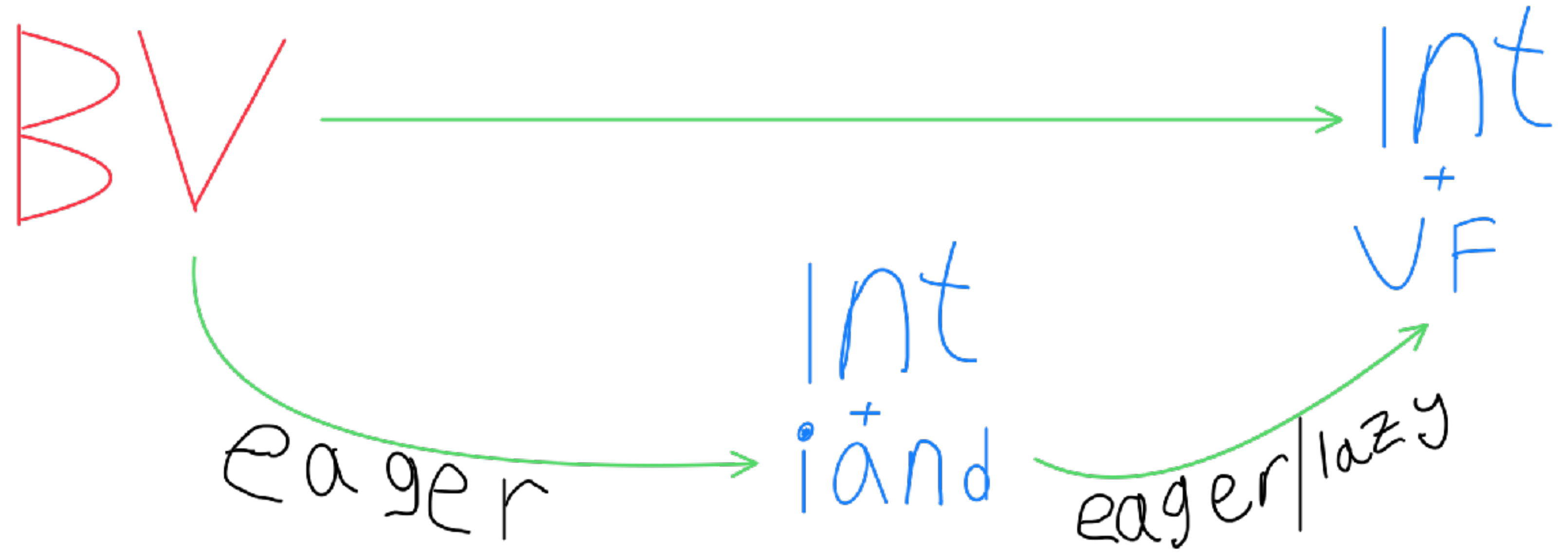
Results — Certora

- Timeout: 1 hour
- Int-blasting solved the most
- Int-blasting was faster:
  - 24 benchmarks in 232 seconds
  - 22 benchmarks in 20 seconds
  - Bitwuzla: 16 benchmarks in 5900 seconds
  - Yices: 9 benchmarks in 3900 seconds

	SMT-LIB				ECRW				SC			
	slvd	*sat*	*uns*	*m*	*slvd*	*sat*	*uns*	*m*	*slvd*	*sat*	*uns*	*m*
$eager_b$	35031	10447	24584	38	41989	119	41870	0	**24**	9	15	0
$eager_s$	35035	10459	24576	28	41435	119	41316	77	**24**	9	15	0
$lazy_b$	35001	10383	24618	23	**47071**	119	46952	0	**24**	9	15	0
$lazy_s$	34819	10297	24522	27	45350	119	45231	138	**24**	9	15	0
Bitwuzla	41220	14233	26987	19	37297	265	37032	11120	16	8	8	0
cvc5	40543	14204	26339	36	33187	220	32967	17535	-	-	-	-
Yices	**41228**	14280	26948	11	31646	255	31391	15801	9	3	6	0
bw-ind	-	-	-	-	25608	0	25608	0	-	-	-	-

Conclusion

- We have seen:
  - Int-blasting is a complement to bit-blasting
  - 4 Configurations (eager/lazy, sum/bitwise)
  - Useful for large bit-widths



- Future Work:
  - Abstraction of other operations
  - More benchmarking
  - Improve non-linear integer solvers

SMT-LIB
THE SATISFIABILITY MODULO THEORIES LIBRARY

Thank You!