From Programs to Formulas

CS294: Program Synthesis for Everyone

Ras Bodik
Emina Torlak

Division of Computer Science
University of California, Berkeley

Today

We show, by example, how to translate a program into formulas that can be used to solve the four programming problems introduced in the last lecture.

- **Reading:** D. Kroening, E. Clarke and K. Yorav. Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking. DAC 2003.

Next lecture: core solver algorithms (by Niklas Eén)

Subsequent lecture: tutorial on the Kodkod solver

Outline

Intro to the SIMD matrix transpose problem

- From imperative code to a functional intermediate form
- From the functional form to formulas

Intro to the theory of integers, arrays and bitvectors

- Encoding transpose using different theories
- Using Racket to generate encodings

HW2: create your own efficient encoding of transpose

Optional reading on SMT, Racket and program encodings

Advanced challenge

Is this lecture familiar material? Entertain yourself by thinking through how to carry out this style of encoding using other theories, e.g.:

- boolean only
- bitvectors only
- bitvectors with uninterpreted functions
- your favorite logic

Example: 4x4-matrix transpose with SIMD

A functional (executable) specification:

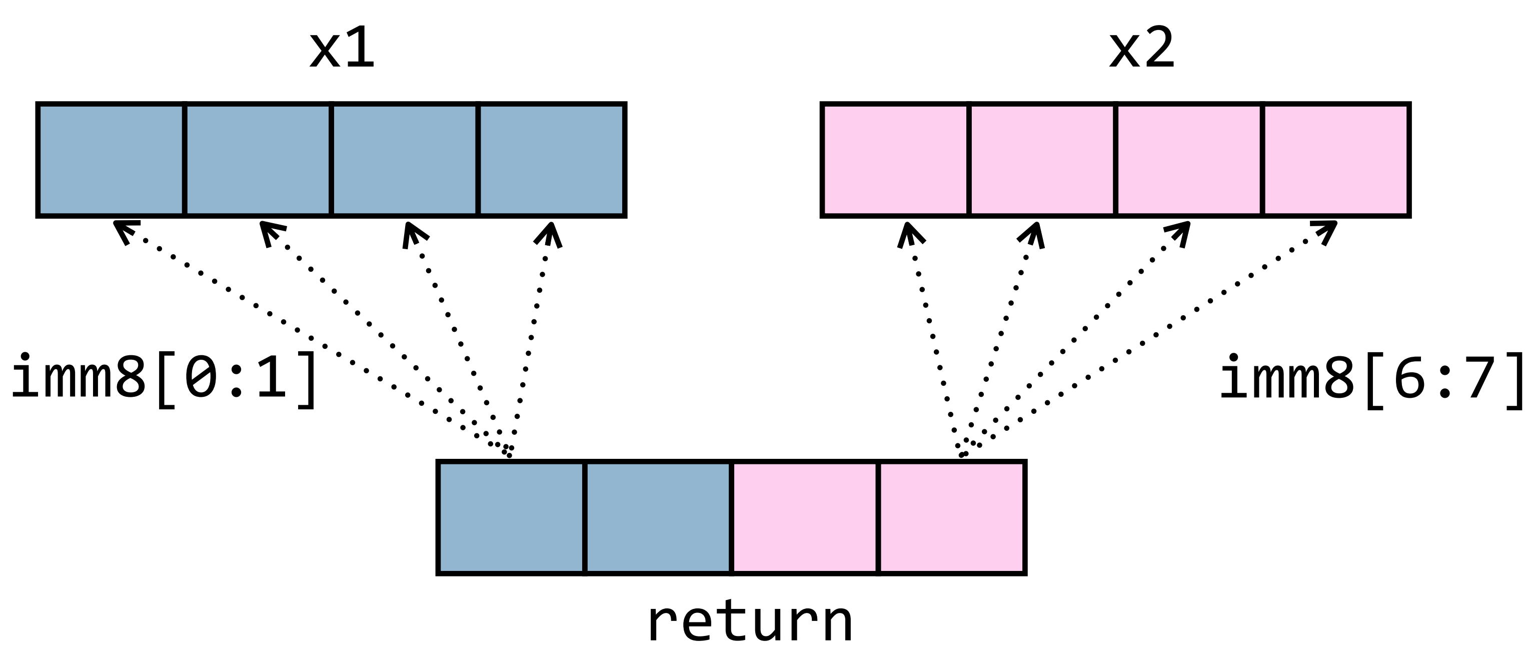
```
int[16] transpose(int[16] M) {
  int[16] T = 0;
  for (int i = 0; i < 4; i++)
    for (int j = 0; j < 4; j++)
      T[4 * i + j] = M[4 * j + i];
  return T;
}
```

This example comes from a Sketch grad-student contest

Implementation idea: parallelize with SIMD

Intel SHUFPS (shuffle parallel scalars) SIMD instruction:

```
return = shufps(x1, x2, imm8 :: bitvector8)
```



High-level insight: transpose as a 2-phase shuffle

Matrix ***M*** can be transposed in two shuffle phases

- **Phase 1:** shuffle ***M*** into an intermediate matrix ***S*** with some number of `shufps` instructions
- **Phase 2:** shuffle ***S*** into the result matrix ***T*** with some number of `shufps` instructions

Synthesis with partial programs helps one to complete their insight. Or prove it wrong.

SIMD matrix transpose, sketched

```
int[16] trans_sse(int[16] M) implements trans {
  int[16] S = 0, T = 0;

  S[??::4] = shufps(M[??::4], M[??::4], ??);
  S[??::4] = shufps(M[??::4], M[??::4], ??);
  …
  S[??::4] = shufps(M[??::4], M[??::4], ??);

  T[??::4] = shufps(S[??::4], S[??::4], ??);
  T[??::4] = shufps(S[??::4], S[??::4], ??);
  …
  T[??::4] = shufps(S[??::4], S[??::4], ??);

  return T;
}
```

Phase 1 (the S[??::4] = shufps(M[…]) lines)

Phase 2 (the T[??::4] = shufps(S[…]) lines)

SIMD matrix transpose, sketched

```
int[16] trans_sse(int[16] M) implements trans {
  int[16] S = 0, T = 0;
  repeat (??) S[??::4] = shufps(M[??::4], M[??::4], ??);
  repeat (??) T[??::4] = shufps(S[??::4], S[??::4], ??);
  return T;
}
```

```
int[16] trans_sse(int[16] M) implements trans { // synthesized code
  S[4::4]   = shufps(M[6::4],   M[2::4],  11001000b);
  S[0::4]   = shufps(M[11::4],  M[6::4],  10010110b);
  S[12::4]  = shufps(M[0::4],   M[2::4],  10001101b);
  S[8::4]   = shufps(M[8::4],   M[12::4], 11010111b);
  T[4::4]   = shufps(S[11::4],  S[1::4],  10111100b);
  T[12::4]  = shufps(S[3::4],   S[8::4],  11000011b);
  T[8::4]   = shufps(S[4::4],   S[9::4],  11100010b);
  T[0::4]   = shufps(S[12::4],  S[0::4],  10110100b);
}
```

SIMD matrix transpose, sketched

```
int[16] trans_sse(int[16] M) implements trans {
  int[16] S = 0, T = 0;
  repeat (??) S[??::4] = shufps(M[??::4], M[??::4], ??);
  repeat (??) T[??::4] = shufps(S[??::4], S[??::4], ??);
  return T;
}
```

From the contestant email:

Over the summer, I spent about 1/2 a day manually figuring it out. *Synthesis time:* < 2 minutes.

```
int[16] trans_sse(int[16] M) implements trans { // synthesized code
  S[4::4]   = shufps(M[6::4],   M[2::4],  11001000b);
  S[0::4]   = shufps(M[11::4],  M[6::4],  10010110b);
  S[12::4]  = shufps(M[0::4],   M[2::4],  10001101b);
  S[8::4]   = shufps(M[8::4],   M[12::4], 11010111b);
  T[4::4]   = shufps(S[11::4],  S[1::4],  10111100b);
  T[12::4]  = shufps(S[3::4],   S[8::4],  11000011b);
  T[8::4]   = shufps(S[4::4],   S[9::4],  11100010b);
  T[0::4]   = shufps(S[12::4],  S[0::4],  10110100b);
}
```

Demo: Sketching SIMD transpose

Try Sketch online at http://bit.ly/sketch-language

Sample sketches of SIMD transpose at CS294/L3/xpose

SIMD matrix transpose with more insight

```
int[16] trans_sse(int[16] M) implements trans {
  int[16] S = 0, T = 0;

  S[??::4] = shufps(M[??::4], M[??::4], ??);
  S[??::4] = shufps(M[??::4], M[??::4], ??);
  S[??::4] = shufps(M[??::4], M[??::4], ??);
  S[??::4] = shufps(M[??::4], M[??::4], ??);

  T[??::4] = shufps(S[??::4], S[??::4], ??);
  T[??::4] = shufps(S[??::4], S[??::4], ??);
  T[??::4] = shufps(S[??::4], S[??::4], ??);
  T[??::4] = shufps(S[??::4], S[??::4], ??);

  return T;
}
```

4 shuffle instructions per phase

SIMD matrix transpose with even more insight

```
int[16] trans_sse(int[16] M) implements trans {
  int[16] S = 0, T = 0;

  S[0::4]  = shufps(M[??::4], M[??::4], ??);
  S[4::4]  = shufps(M[??::4], M[??::4], ??);
  S[8::4]  = shufps(M[??::4], M[??::4], ??);
  S[12::4] = shufps(M[??::4], M[??::4], ??);

  T[0::4]  = shufps(S[??::4], S[??::4], ??);
  T[4::4]  = shufps(S[??::4], S[??::4], ??);
  T[8::4]  = shufps(S[??::4], S[??::4], ??);
  T[12::4] = shufps(S[??::4], S[??::4], ??);

  return T;
}
```

1 shuffle instruction per row of output

From SIMD transpose to formulas in 4 steps

From SIMD transpose to formulas in 4 steps

1. Make correctness conditions and synthesis constructs explicit
   - `implements` construct becomes an assertion
   - each hole `??` becomes a fresh symbolic variable

From SIMD transpose to formulas in 4 steps

1. Make correctness conditions and synthesis constructs explicit
   - `implements` construct becomes an assertion
   - each hole `??` becomes a fresh symbolic variable
2. Unroll loops to obtain a bounded acyclic program
   - see next week's reading for details

From SIMD transpose to formulas in 4 steps

1. Make correctness conditions and synthesis constructs explicit
   - `implements` construct becomes an assertion
   - each hole `??` becomes a fresh symbolic variable
2. Unroll loops to obtain a bounded acyclic program
   - see next week's reading for details
3. Make the resulting straight-line code functional
   - use SSA to eliminate side effects

From SIMD transpose to formulas in 4 steps

1. Make correctness conditions and synthesis constructs explicit
   – `implements` construct becomes an assertion
   – each hole `??` becomes a fresh symbolic variable
2. Unroll loops to obtain a bounded acyclic program
   – see next week's reading for details
3. Make the resulting straight-line code functional
   – use SSA to eliminate side effects
4. "Read off" a formula from this functional program
   – map the program's operational semantics into a logic
   – we'll look at the theories of integers, arrays and bitvectors

From SIMD transpose to formulas (step 1)

```
int[16] trans_sse(int[16] M) implements trans {
  int[16] S = 0, T = 0;
  S[0::4]  = shufps(M[??::4], M[??::4], ??);
  S[4::4]  = shufps(M[??::4], M[??::4], ??);
  S[8::4]  = shufps(M[??::4], M[??::4], ??);
  S[12::4] = shufps(M[??::4], M[??::4], ??);
  T[0::4]  = shufps(S[??::4], S[??::4], ??);
  T[4::4]  = shufps(S[??::4], S[??::4], ??);
  T[8::4]  = shufps(S[??::4], S[??::4], ??);
  T[12::4] = shufps(S[??::4], S[??::4], ??);
  return T;
}
```

From SIMD transpose to formulas (step 1)

```
int[16] trans_sse(int[16] M) {
  int[16] S = 0, T = 0;
  S[0::4]  = shufps(M[??::4], M[??::4], ??);
  S[4::4]  = shufps(M[??::4], M[??::4], ??);
  S[8::4]  = shufps(M[??::4], M[??::4], ??);
  S[12::4] = shufps(M[??::4], M[??::4], ??);
  T[0::4]  = shufps(S[??::4], S[??::4], ??);
  T[4::4]  = shufps(S[??::4], S[??::4], ??);
  T[8::4]  = shufps(S[??::4], S[??::4], ??);
  T[12::4] = shufps(S[??::4], S[??::4], ??);
  assert equals(T, trans(M));
  return T;
}
```

Make the correctness condition explicit: trans_sse implements trans

From SIMD transpose to formulas (step 1)

```
int[16] trans_sse(int[16] M) {
  int[16] S = 0, T = 0;
  S[0::4]  = shufps(M[mx1_0::4], M[mx2_0::4], mi_0);
  S[4::4]  = shufps(M[mx1_1::4], M[mx2_1::4], mi_1);
  S[8::4]  = shufps(M[mx1_2::4], M[mx2_2::4], mi_2);
  S[12::4] = shufps(M[mx1_3::4], M[mx2_3::4], mi_3);
  T[0::4]  = shufps(S[sx1_0::4], S[sx2_0::4], si_0);
  T[4::4]  = shufps(S[sx1_1::4], S[sx2_1::4], si_1);
  T[8::4]  = shufps(S[sx1_2::4], S[sx2_2::4], si_2);
  T[12::4] = shufps(S[sx1_3::4], S[sx2_3::4], si_3);
  assert equals(T, trans(M));
  return T;
}
```

Name the holes: each corresponds to a fresh symbolic variable.

From SIMD transpose to formulas (step 3)

Turn bulk array accesses into explicit calls to a read function.

```
int[16] trans_sse(int[16] M) {
  int[16] S = 0, T = 0;
  S[0::4]  = shufps(rd4(M, mx1_0), rd4(M, mx2_0), mi_0);
  S[4::4]  = shufps(rd4(M, mx1_1), rd4(M, mx2_1), mi_1);
  S[8::4]  = shufps(rd4(M, mx1_2), rd4(M, mx2_2), mi_2);
  S[12::4] = shufps(rd4(M, mx1_3), rd4(M, mx2_3), mi_3);
  T[0::4]  = shufps(rd4(S, sx1_0), rd4(S, sx2_0), si_0);
  T[0::4]  = shufps(rd4(S, sx1_1), rd4(S, sx2_1), si_1);
  T[0::4]  = shufps(rd4(S, sx1_2), rd4(S, sx2_2), si_2);
  T[0::4]  = shufps(rd4(S, sx1_3), rd4(S, sx2_3), si_3);
  assert equals(T, trans(M));
  return T;
}
```

rd4(A, i) returns a new array consisting of A[i], ..., A[i+3].

From SIMD transpose to formulas (step 3)

Convert to SSA by replacing bulk array writes with functional writes.

```
int[16] trans_sse(int[16] M) {
  int[16] S = 0, T = 0;
  S₀ = wr4(S , shufps(rd4(M, mx1_0), rd4(M, mx2_0), mi_0), 0);
  S₁ = wr4(S₀, shufps(rd4(M, mx1_1), rd4(M, mx2_1), mi_1), 4);
  S₂ = wr4(S₁, shufps(rd4(M, mx1_2), rd4(M, mx2_2), mi_2), 8);
  S₃ = wr4(S₂, shufps(rd4(M, mx1_3), rd4(M, mx2_3), mi_3), 12);
  T₀ = wr4(T , shufps(rd4(S₃, sx1_0), rd4(S₃, sx2_0), si_0), 0);
  T₁ = wr4(T₀, shufps(rd4(S₃, sx1_1), rd4(S₃, sx2_1), si_1), 4);
  T₂ = wr4(T₁, shufps(rd4(S₃, sx1_2), rd4(S₃, sx2_2), si_2), 8);
  T₃ = wr4(T₂, shufps(rd4(S₃, sx1_3), rd4(S₃, sx2_3), si_3), 12);
  assert equals(T₃, trans(M));
  return T₃;
}
```

wr4(A, Delta, i) returns a copy of A, but with Delta[0::4] at positions i, ..., i+3.

From SIMD transpose to formulas (step 4)

Once the program is functional, turn it into a formula.

- Many encodings of programs as formulas are possible.
- Some encodings are faster to solve than others.

Times from our experiments with encoding transpose:

encoding	solver	time (sec)*
QF_AUFLIA	CVC3	>600
	Z3	159
QF_AUFBV	Boolector	409
	Z3	287
	CVC3	119
QF_AUFBV (non extensional)	CVC3	>600
	Boolector	>600
	Z3	25
	STP	11
REL_BV	Rosette	9
REL	Kodkod	5

*MacBook Air, 2.13 GHz Intel Core 2 Duo, 4 GB RAM, OS X 10.7.4

From SIMD transpose to formulas (step 4)

Once the program is functional, turn it into a formula.

- Many encodings of programs as formulas are possible.
- Some encodings are faster to solve than others.

Times from our experiments with encoding transpose:

encoding	solver	time (sec)*
QF_AUFLIA	CVC3	>600
	Z3	159
QF_AUFBV	Boolector	409
	Z3	287
	CVC3	119
QF_AUFBV (non extensional)	CVC3	>600
	Boolector	>600
	Z3	25
	STP	11

*MacBook Air, 2.13 GHz Intel Core 2 Duo, 4 GB RAM, OS X 10.7.4

From SIMD transpose to Z3 formulas

From SIMD transpose to Z3 formulas

Z3 input language is a superset of the SMT-LIB 2.0 standard

- we make use of some Z3-specific features for brevity, e.g.:
  - constant arrays (in this lecture)
  - algebraic datatypes (in the last lecture)
  - relations and fixed point constraints (datalog, in your project?)
- other solvers require SMT-LIB 2.0 encodings that are more verbose (fully expanded) versions of the Z3 encodings

From SIMD transpose to Z3 formulas

Z3 input language is a superset of the SMT-LIB 2.0 standard

- we make use of some Z3-specific features for brevity, e.g.:
  - constant arrays (in this lecture)
  - algebraic datatypes (in the last lecture)
  - relations and fixed point constraints (datalog, in your project?)
- other solvers require SMT-LIB 2.0 encodings that are more verbose (fully expanded) versions of the Z3 encodings

Z3 supports a rich set of **theories**

- e.g., arrays, integers, bitvectors, reals, and many others
- informally: a theory describes the set of all formulas that we can write using a given set of types (*sorts*) and operations (*functions*) over those types, chosen to be efficiently solvable
- for formal definitions, see Leonardo De Moura's SMT tutorial

From SIMD transpose to Z3 integers & arrays

```
int[16] trans_sse(int[16] M) {
  int[16] S = 0, T = 0;
  S0 = wr4(S , shufps(rd4(M, mx1_0), rd4(M, mx2_0), mi_0), 0);
  S1 = wr4(S0, shufps(rd4(M, mx1_1), rd4(M, mx2_1), mi_1), 4);
  S2 = wr4(S1, shufps(rd4(M, mx1_2), rd4(M, mx2_2), mi_2), 8);
  S3 = wr4(S2, shufps(rd4(M, mx1_3), rd4(M, mx2_3), mi_3), 12);
  T0 = wr4(T , shufps(rd4(S3, sx1_0), rd4(S3, sx2_0), si_0), 0);
  T1 = wr4(T0, shufps(rd4(S3, sx1_1), rd4(S3, sx2_1), si_1), 4);
  T2 = wr4(T1, shufps(rd4(S3, sx1_2), rd4(S3, sx2_2), si_2), 8);
  T3 = wr4(T2, shufps(rd4(S3, sx1_3), rd4(S3, sx2_3), si_3), 12);
  assert equals(T3, trans(M));
  return T3;
}
```

From SIMD transpose to Z3 integers & arrays

Encode mx1, mx2, mi, sx1, sx2 and si holes as integer variables.

```
int[16] trans_sse(int[16] M) {
  int[16] S = 0, T = 0;
  S0 = wr4(S , shufps(rd4(M, mx1_0), rd4(M, mx2_0), mi_0), 0);
  S1 = wr4(S0, shufps(rd4(M, mx1_1), rd4(M, mx2_1), mi_1), 4);
  S2 = wr4(S1, shufps(rd4(M, mx1_2), rd4(M, mx2_2), mi_2), 8);
  S3 = wr4(S2, shufps(rd4(M, mx1_3), rd4(M, mx2_3), mi_3), 12);
  T0 = wr4(T , shufps(rd4(S3, sx1_0), rd4(S3, sx2_0), si_0), 0);
  T1 = wr4(T0, shufps(rd4(S3, sx1_1), rd4(S3, sx2_1), si_1), 4);
  T2 = wr4(T1, shufps(rd4(S3, sx1_2), rd4(S3, sx2_2), si_2), 8);
  T3 = wr4(T2, shufps(rd4(S3, sx1_3), rd4(S3, sx2_3), si_3), 12);
  assert equals(T3, trans(M));
  return T3;
}
```

From SIMD transpose to Z3 integers & arrays

Encode mx1, mx2, mi, sx1, sx2 and si holes as integer variables.

```
int[16] trans_sse(int[16] M) {
  int[16] S = 0, T = 0;
  S₀ = wr4(S , shufps(rd4(M, mx1_0), rd4(M, mx2_0), mi_0), 0);
  S₁ = wr4(S₀, shufps(rd4(M, mx1_1), rd4(M, mx2_1), mi_1), 4);
  S₂ = wr4(S₁, shufps(rd4(M, mx1_2), rd4(M, mx2_2), mi_2), 8);
  S₃ = wr4(S₂, shufps(rd4(M, mx1_3), rd4(M, mx2_3), mi_3), 12);
  T₀ = wr4(T , shufps(rd4(S₃, sx1_0), rd4(S₃, sx2_0), si_0), 0);
  T₁ = wr4(T₀, shufps(rd4(S₃, sx1_1), rd4(S₃, sx2_1), si_1), 4);
  T₂ = wr4(T₁, shufps(rd4(S₃, sx1_2), rd4(S₃, sx2_2), si_2), 8);
  T₃ = wr4(T₂, shufps(rd4(S₃, sx1_3), rd4(S₃, sx2_3), si_3), 12);
  assert equals(T₃, trans(M));
  return T₃;
}
```

Encode M, trans(M), S_0 and T_0 as arrays variables with integer indices and value.

From SIMD transpose to Z3 integers & arrays

```
; an mx1_j hole is an integer in [0..12]
(declare-const mx1_0 Int)
(assert (and (<= 0 mx1_0) (<= mx1_0 12)))
```

declare-const introduces a variable of a given type, or *sort*.

assert adds a formula to the solver's internal stack.

From SIMD transpose to Z3 integers & arrays

```
; an mx1_j hole is an integer in [0..12]
(declare-const mx1_0 Int)
(assert (and (<= 0 mx1_0) (<= mx1_0 12)))
```

declare-const introduces a variable of a given type, or *sort*.

assert adds a formula to the solver's internal stack.

Where does this formula come from?

From SIMD transpose to Z3 integers & arrays

```
; an mx1_j hole is an integer in [0..12]
(declare-const mx1_0 Int)
(assert (and (<= 0 mx1_0) (<= mx1_0 12)))
```

declare-const introduces a variable of a given type, or *sort*.

assert adds a formula to the solver's internal stack.

Language semantics: array access `rd4(M, mx1_0)` must be within bounds, so mx1_j ∈ [0 .. length(M) - 4] = [0 .. 16 - 4] = [0 .. 12].

From SIMD transpose to Z3 integers & arrays

```
; an mx1_j hole is an integer in [0..12]
(declare-const mx1_0 Int)
(assert (and (<= 0 mx1_0) (<= mx1_0 12)))

; an mi_j hole holds 4 integers in [0..3]
(declare-const mi_0_0 Int)
(declare-const mi_0_1 Int)
(declare-const mi_0_2 Int)
(declare-const mi_0_3 Int)

(assert (and (<= 0 mi_0_0) (<= mi_0_0 3)))
(assert (and (<= 0 mi_0_1) (<= mi_0_1 3)))
(assert (and (<= 0 mi_0_2) (<= mi_0_2 3)))
(assert (and (<= 0 mi_0_3) (<= mi_0_3 3)))
```

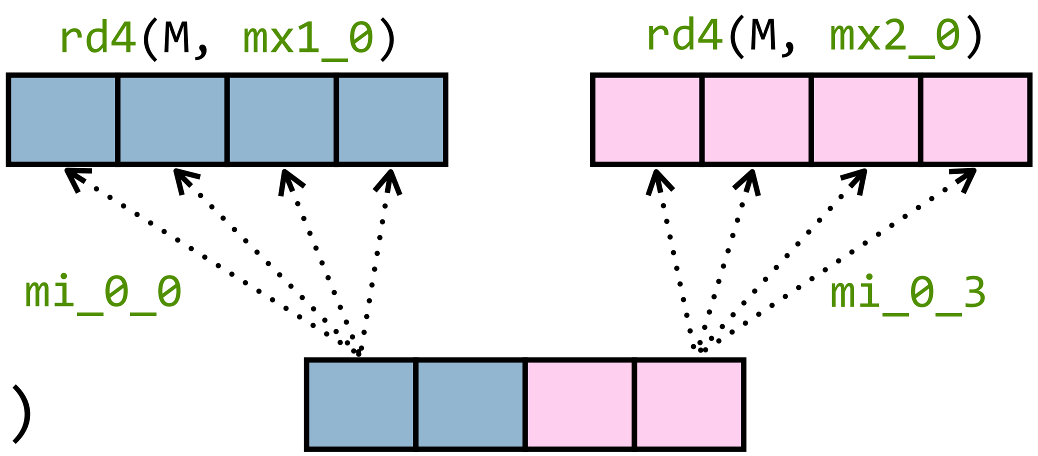
From SIMD transpose to Z3 integers & arrays

```
; an mx1_j hole is an integer in [0..12]
(declare-const mx1_0 Int)
(assert (and (<= 0 mx1_0) (<= mx1_0 12)))

; an mi_j hole holds 4 integers in [0..3]
(declare-const mi_0_0 Int)
(declare-const mi_0_1 Int)
(declare-const mi_0_2 Int)
(declare-const mi_0_3 Int)

(assert (and (<= 0 mi_0_0) (<= mi_0_0 3)))
(assert (and (<= 0 mi_0_1) (<= mi_0_1 3)))
(assert (and (<= 0 mi_0_2) (<= mi_0_2 3)))
(assert (and (<= 0 mi_0_3) (<= mi_0_3 3)))
```

Recall from the definition of shufps that mi_j is an 8-bit value, interpreted as four 2-bit values.



From SIMD transpose to Z3 integers & arrays

```
#lang racket

(define var
  (case-lambda [(base i) (string->symbol (format "~a_~a" base i))]
               [(base i j) (string->symbol (format "~a_~a_~a" base i j))]))

(define (declare-int-const v min max)
  (pretty-display `(declare-const ,v Int))
  (pretty-display `(assert (and (<= ,min ,v) (<= ,v ,max)))))

(define (declare-int-consts)
  (for* ([v `(mx1 mx2 sx1 sx2)]
         [i (in-range 0 4)])
    (declare-int-const (var v i) 0 12))
  (for* ([v `(mi si)]
         [i (in-range 0 4)]
         [j (in-range 0 4)])
    (declare-int-const (var v i j) 0 3)))
```

Generate the encoding with Racket.

From SIMD transpose to Z3 integers & arrays

```
; a sample input m: [0, 1, …, 15]
(declare-const m (Array Int Int))
(assert (= 0  (select m 0)))
(assert (= 1  (select m 1)))
…
```

(Array I V) introduces an array sort with indices of sort **I** and values of sort **V**.

(select a i) returns the value stored at the position **i** of the array **a.**

From SIMD transpose to Z3 integers & arrays

```
; a sample input m: [0, 1, …, 15]
(declare-const m (Array Int Int))
(assert (= 0  (select m 0)))
(assert (= 1  (select m 1)))
…

; mt = trans(M) is the transpose of m
(declare-const mt (Array Int Int))
(assert (= 0  (select mt 0)))
(assert (= 1  (select mt 4)))
…

; S0 and T0 are initially empty
(define-fun s () (Array Int Int)
    ((as const (Array Int Int)) 0))
(define-fun t () (Array Int Int)
    ((as const (Array Int Int)) 0))
```

((as const (Array I V)) v) defines a constant array that maps all indices to the value **v.**

From SIMD transpose to Z3 integers & arrays

```
; rd4(a, i) returns a new array consisting of a[i], ..., a[i+3]
(define-fun rd4
 ((a (Array Int Int)) (i Int)) (Array Int Int)
 (store (store (store (store ((as const (Array Int Int)) 0)
     0 (select a i))
     1 (select a (+ i 1)))
     2 (select a (+ i 2)))
     3 (select a (+ i 3))))
```

(store a i v) returns a new array that is identical to **a,** except that it stores **v** at position **i.**

From SIMD transpose to Z3 integers & arrays

```
; rd4(a, i) returns a new array consisting of a[i], …, a[i+3]
(define-fun rd4 ((a (Array Int Int)) (i Int)) (Array Int Int)
 (store (store (store (store ((as const (Array Int Int)) 0)
     0 (select a i))
     1 (select a (+ i 1)))
     2 (select a (+ i 2)))
     3 (select a (+ i 3))))

; wr4(a, d, i) returns a copy of a, but with d[0::4]
; at positions i, …, i+3.
(define-fun wr4 ((a (Array Int Int)) (d (Array Int Int)) (i Int))
                (Array Int Int)
 (store (store (store (store a
     i       (select d 0))
     (+ i 1) (select d 1))
     (+ i 2) (select d 2))
     (+ i 3) (select d 3)))
```

From SIMD transpose to Z3 integers & arrays

```
(define-fun shufps
 ((xmm1 (Array Int Int)) (xmm2 (Array Int Int))
  (imm8_0 Int) (imm8_1 Int) (imm8_2 Int) (imm8_3 Int))
 (Array Int Int)
 (store (store (store (store ((as const (Array Int Int)) 0)
     0 (select xmm1 imm8_0))
     1 (select xmm1 imm8_1))
     2 (select xmm2 imm8_2))
     3 (select xmm2 imm8_3)))
```

From SIMD transpose to Z3 integers & arrays

```
(define-fun shufps
 ((xmm1 (Array Int Int)) (xmm2 (Array Int Int))
  (imm8_0 Int) (imm8_1 Int) (imm8_2 Int) (imm8_3 Int))
 (Array Int Int)
 (store (store (store (store ((as const (Array Int Int)) 0)
     0 (select xmm1 imm8_0))
     1 (select xmm1 imm8_1))
     2 (select xmm2 imm8_2))
     3 (select xmm2 imm8_3)))

; S0 = wr4(S , shufps(rd4(M, mx1_0), rd4(M, mx2_0), mi_0), 0)
(define-fun s0 () (Array Int Int)
 (wr4 s (shufps (rd4 m mx1_0) (rd4 m mx2_0)
                mi_0_0 mi_0_1 mi_0_2 mi_0_3) 0))
…
```

From SIMD transpose to Z3 integers & arrays

```
(define-fun shufps
 ((xmm1 (Array Int Int)) (xmm2 (Array Int Int))
  (imm8_0 Int) (imm8_1 Int) (imm8_2 Int) (imm8_3 Int))
 (Array Int Int)
 (store (store (store (store ((as const (Array Int Int)) 0)
     0 (select xmm1 imm8_0))
     1 (select xmm1 imm8_1))
     2 (select xmm2 imm8_2))
     3 (select xmm2 imm8_3)))

; S0 = wr4(S , shufps(rd4(M, mx1_0), rd4(M, mx2_0), mi_0), 0)
(define-fun s0 () (Array Int Int)
 (wr4 s (shufps (rd4 m mx1_0) (rd4 m mx2_0)
                mi_0_0 mi_0_1 mi_0_2 mi_0_3) 0))
…

(assert (= t3 mt))
```

Theory of arrays is **extensional** over select: the solver enforces equality between every pair of arrays that agree on all reads.

From SIMD transpose to Z3 bitvectors & arrays

```
(define-sort BV2 () (_ BitVec 2))
(define-sort BV4 () (_ BitVec 4))

; an mx1_j hole is a 4-bit value in [0..12]
(declare-const mx1_0 BV4)
(assert (and (bvule (_ bv0 4) mx1_0)
             (bvule mx1_0 (_ bv12 4))))

; an mi_j hole holds four 2-bit values in [0..3]
(declare-const mi_0_0 BV2)
```

From SIMD transpose to Z3 bitvectors & arrays

```
(define-sort BV2 () (_ BitVec 2))
(define-sort BV4 () (_ BitVec 4))
```

define-sort introduces a name for a sort; **(_ BitVec k)** is the sort of bitvectors of length **k.**

```
; an mx1_j hole is a 4-bit value in [0..12]
(declare-const mx1_0 BV4)
(assert (and (bvule (_ bv0 4) mx1_0)
             (bvule mx1_0 (_ bv12 4))))

; an mi_j hole holds four 2-bit values in [0..3]
(declare-const mi_0_0 BV2)
```

From SIMD transpose to Z3 bitvectors & arrays

```
(define-sort BV2 () (_ BitVec 2))
(define-sort BV4 () (_ BitVec 4))
```

define-sort introduces a name for a sort; **(_ BitVec k)** is the sort of bitvectors of length **k.**

```
; an mx1_j hole is a 4-bit value in [0..12]
(declare-const mx1_0 BV4)
(assert (and (bvule (_ bv0 4) mx1_0)
             (bvule mx1_0 (_ bv12 4))))
```

(_ bv*V* *n*) returns a bitvector value ***V*** of length ***n***

```
; an mi_j hole holds four 2-bit values in [0..3]
(declare-const mi_0_0 BV2)
```

From SIMD transpose to Z3 bitvectors & arrays

```
(define-sort BV2 () (_ BitVec 2))
(define-sort BV4 () (_ BitVec 4))
```

define-sort introduces a name for a sort; **(_ BitVec k)** is the sort of bitvectors of length **k.**

```
; an mx1_j hole is a 4-bit value in [0..12]
(declare-const mx1_0 BV4)
(assert (and (bvule (_ bv0 4) mx1_0)
             (bvule mx1_0 (_ bv12 4))))
```

(_ bv*V* *n*) returns a bitvector value ***V*** of length ***n***

```
; an mi_j hole holds four 2-bit values in [0..3]
(declare-const mi_0_0 BV2)
```

Bitvectors are unsigned, so there is no need to assert that mi_0_0 is in [0 .. 3].

From SIMD transpose to Z3 bitvectors & arrays

```
(define-sort BV2 () (_ BitVec 2))
(define-sort BV4 () (_ BitVec 4))
```

define-sort introduces a name for a sort; **(_ BitVec k)** is the sort of bitvectors of length **k.**

```
; an mx1_j hole is a 4-bit value in [0..12]
(declare-const mx1_0 BV4)
(assert (and (bvule (_ bv0 4) mx1_0)
             (bvule mx1_0 (_ bv12 4))))
```

(_ bv*V* *n*) returns a bitvector value ***V*** of length ***n***

```
; an mi_j hole holds four 2-bit values in [0..3]
(declare-const mi_0_0 BV2)
```

Bitvectors are unsigned, so there is no need to assert that mi_0_0 is in [0 .. 3].

Why use bitvectors?

From SIMD transpose to Z3 bitvectors & arrays

```
(define-sort BV2 () (_ BitVec 2))
(define-sort BV4 () (_ BitVec 4))
```

define-sort introduces a name for a sort; **(_ BitVec k)** is the sort of bitvectors of length **k.**

```
; an mx1_j hole is a 4-bit value in [0..12]
(declare-const mx1_0 BV4)
(assert (and (bvule (_ bv0 4) mx1_0)
             (bvule mx1_0 (_ bv12 4))))
```

(_ bv*V* *n*) returns a bitvector value ***V*** of length ***n***

```
; an mi_j hole holds four 2-bit values in [0..3]
(declare-const mi_0_0 BV2)
```

Bitvectors are unsigned, so there is no need to assert that mi_0_0 is in [0 .. 3].

Why use bitvectors?

- Precise modeling of machine arithmetic
- Decided by **bit-blasting,** which can be more efficient than Simplex (for integers)

HW2: An Efficient Encoding of Transpose

Part 1

- Complete the QF_AUFBV encoding of SIMD transpose
- The resulting should be significantly faster than QF_AUFLIA
- Hint: use non-extensional theory of arrays

Part 2

- Create an encoding for SIMD transpose with unknowns on both the left and the right hand side (slide 11)

Extra credit

- Scale your encoding to larger matrices: 8x8, 16x16, etc.
- Try a different solver
- Try an encoding that does not correspond to the operational semantics of transpose

References: SMT

SMT-LIB language and benchmarks

- Clark Barrett, Aaron Stump and Cesare Tinelli. The SMT-LIB Standard Version 2.0, 2010.
- David R. Cok, The SMT-LIB v2 Language and Tools: A Tutorial, 2012.
- SMT-COMP (find the best solver for your problem)

Overview of SMT terminology and approaches

- Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability Modulo Theories. In Armin Biere, Hans van Maaren, and Toby Walsh, editors, Handbook of Satisfiability, IOS Press, 2009.
- Leonardo de Moura. SMT Solvers: Theory and Implementation. *Summer School on Logic and Theorem Proving*, Oregon, 2008.

SMT solvers

- Solvers used in the SIMD transpose experiments: Boolector, CVC3, STP, Z3.
- Getting Started with Z3: A Guide. http://rise4fun.com/z3/tutorial/guide, 2012.

References: Racket

A selection of Racket tutorials, tools, and documentation

– Matthew Flatt, Quick: An Introduction to Racket with Pictures.
– Matthew Flatt, Robert Findler, and PLT. The Racket Guide.
– Matthew Flatt and PLT. The Racket Reference.
– Noel Welsh and Ryan Culpepper. RackUnit: Unit Testing.
– Profile: Statistical Profiler.

Two fun, easy to read guides to embedding languages in Racket

– Danny Yoo. F*dging up a Racket.
– Matthew Flatt. Creating Languages in Racket. *CACM*, Vol. 55 No. 1, Pages 48-56, Jan. 2012.

References: program encodings (a tiny sample)

Bounded model checking of sequential programs using SAT/SMT

- Julian Dolby, Mandana Vaziri, and Frank Tip. Finding bugs efficiently with a SAT solver. FSE 2007.
- Daniel Kroening, Edmund Clarke and Karen Yorav. Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking. DAC 2003.
- Yichen Xie and Alex Aiken. Scalable Error Detection using Boolean Satisfiability. POPL 2005.

Bounded model checking of concurrent programs using SAT/SMT

- Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. CheckFence: checking consistency of concurrent data types on relaxed memory models. PLDI 2007.
- Akash Lal and Thomas Reps, Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis, FMSD 2009.
- Ishai Rabinovitz and Orna Grumberg. Bounded model checking of concurrent programs. CAV 2005.
- Emina Torlak, Mandana Vaziri, and Julian Dolby. MemSAT: checking axiomatic specifications of memory models. PLDI 2010.

SMT-based verification (no bounding)

- K. Rustan M. Leino and Philipp Rümmer. A Polymorphic Intermediate Verification Language: Design and Logical Encoding. TACAS 2010.
- K. Rustan M. Leino, Peter Müller, and Jan Smans. Verification of Concurrent Programs with Chalice. FOSAD 2009.