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.


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:

```c
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:

\[ \text{return} = \text{shufps}(x1, x2, \text{imm8} :: \text{bitvector8}) \]
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 shuffle instructions
- **Phase 2:** shuffle $S$ into the result matrix $T$ with some number of shuffle instructions

Synthesis with partial programs helps one to complete their insight. Or prove it wrong.
SIMD matrix transpose, sketched

```c
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

Phase 2
The name of the course this CS294 topics course has been listed as CS294: Programming Language Design for Everyone. Since putting the course on the books, we realized we are ready to teach a superset of intended material.

In addition to-

- design of domain-specific languages (DSLs) and their lightweight implementation,

you will learn-

- how to build a synthesizer in a semester also a topic for everyone (PL students and others).

SIMD matrix transpose, sketched

```c
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);
    return T;
}
```
The name of the course this CS294 topics course has been listed as CS294: Programming Language Design for Everyone. Since putting the course on the books, we realized we are ready to teach a superset of intended material.

In addition to:
- design of domain-specific languages (DSLs) and their lightweight implementation,

you will learn:
- how to build a synthesizer in a semester.

Also a topic for everyone (PL students and others).

SIMD matrix transpose, sketched

```c
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.

```c
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
The name of the course this CS294 topics course has been listed as CS294: Programming Language Design for Everyone. Since putting the course on the books, we realized we are ready to teach a superset of intended material.

In addition to-
- design of domain-specific languages (DSLs) and their lightweight implementation

you will learn-
- how to build a synthesizer in a semester

also a topic for everyone (PL students and others)

SIMD matrix transpose with more insight

```c
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
The name of the course this CS294 topics course has been listed as CS294: Programming Language Design for Everyone. Since putting the course on the books, we are ready to teach a superset of intended material. In addition to:

- design of domain-specific languages (DSLs) and
- their lightweight implementation

you will learn:

- how to build a synthesizer in a semester

also a topic for everyone (PL students and others)

SIMD matrix transpose with even more insight

```c
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
   - `impl`emented 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
The name of the course this CS294 topics course has been listed as CS294: Programming Language Design for Everyone. Since putting the course on the books, we are ready to teach a superset of intended material.

In addition to:

- design of domain-specific languages (DSLs) and their lightweight implementation
- you will learn how to build a synthesizer in a semester also a topic for everyone (PL students and others).

From SIMD transpose to formulas in 4 steps:

1. Make correctness conditions and synthesis constructs explicit
   - **impl**ement 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)

```plaintext
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;
}
```
The name of the course this CS294 topics course has been listed as CS294: Programming Language Design for Everyone. Since putting the course on the books, we are ready to teach a superset of intended material. In addition to:

- design of domain-specific languages (DSLs) and
- their lightweight implementation

you will learn:

- how to build a synthesizer in a semester

also a topic for everyone (PL students and others)

From SIMD transpose to formulas (step 1)

```c
int[16] trans_sse(int[16] M) {
    int[16] S = 0, T = 0;
    S[0::4] = shufps(M[0::4], M[1::4], 0);
    S[4::4] = shufps(M[1::4], M[2::4], 0);
    S[8::4] = shufps(M[2::4], M[3::4], 0);
    S[12::4] = shufps(M[3::4], M[4::4], 0);
    T[0::4] = shufps(S[0::4], S[1::4], 0);
    T[4::4] = shufps(S[1::4], S[2::4], 0);
    T[8::4] = shufps(S[2::4], S[3::4], 0);
    T[12::4] = shufps(S[3::4], S[4::4], 0);
    assert equals(T, trans(M));
    return T;
}
```

Make the correctness condition explicit: trans_sse implements trans
From SIMD transpose to formulas (step 1)

```c
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.
The name of the course this CS294 topics course has been listed as CS294: Programming Language Design for Everyone.

Since putting the course on the books, we realized we are ready to teach a superset of intended material. In addition to:

- design of domain-specific languages (DSLs) and their lightweight implementation,
- you will learn how to build a synthesizer in a semester,

also a topic for everyone (PL students and others).

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

```c
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;
}
```

The name of the course this CS294 topics course has been listed as CS294: Programming Language Design for Everyone. Since putting the course on the books, we realized we are ready to teach a superset of intended material. In addition to design of domain-specific languages (DSLs) and their lightweight implementation, you will learn how to build a synthesizer in a semester also a topic for everyone (PL students and others).

From SIMD transpose to formulas (step 3)

```
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;
}
```

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

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:

<table>
<thead>
<tr>
<th>encoding</th>
<th>solver</th>
<th>time (sec)*</th>
</tr>
</thead>
<tbody>
<tr>
<td>QF_AUFLIA</td>
<td>CVC3</td>
<td>&gt;600</td>
</tr>
<tr>
<td></td>
<td>Z3</td>
<td>159</td>
</tr>
<tr>
<td>QF_AUFBV</td>
<td>Boolector</td>
<td>409</td>
</tr>
<tr>
<td></td>
<td>Z3</td>
<td>287</td>
</tr>
<tr>
<td></td>
<td>CVC3</td>
<td>119</td>
</tr>
<tr>
<td>QF_AUFBV (non extensional)</td>
<td>CVC3</td>
<td>&gt;600</td>
</tr>
<tr>
<td></td>
<td>Boolector</td>
<td>&gt;600</td>
</tr>
<tr>
<td></td>
<td>Z3</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>STP</td>
<td>11</td>
</tr>
<tr>
<td>REL_BV</td>
<td>Rosette</td>
<td>9</td>
</tr>
<tr>
<td>REL</td>
<td>Kodkod</td>
<td>5</td>
</tr>
</tbody>
</table>

*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:

<table>
<thead>
<tr>
<th>encoding</th>
<th>solver</th>
<th>time (sec)*</th>
</tr>
</thead>
<tbody>
<tr>
<td>QF_AUFLIA</td>
<td>CVC3</td>
<td>&gt;600</td>
</tr>
<tr>
<td></td>
<td>Z3</td>
<td>159</td>
</tr>
<tr>
<td>QF_AUFBV</td>
<td>Boolector</td>
<td>409</td>
</tr>
<tr>
<td></td>
<td>Z3</td>
<td>287</td>
</tr>
<tr>
<td></td>
<td>CVC3</td>
<td>119</td>
</tr>
<tr>
<td>QF_AUFBV (non extensional)</td>
<td>CVC3</td>
<td>&gt;600</td>
</tr>
<tr>
<td></td>
<td>Boolector</td>
<td>&gt;600</td>
</tr>
<tr>
<td></td>
<td>Z3</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>STP</td>
<td>11</td>
</tr>
</tbody>
</table>

*MacBook Air, 2.13 GHz Intel Core 2 Duo, 4 GB RAM, OS X 10.7.4
The name of the course this CS294 topics course has been listed as CS294: Programming Language Design for Everyone. Since putting the course on the books, we realized we are ready to teach a superset of intended material. In addition to:
- design of domain-specific languages (DSLs) and their lightweight implementation,

you will learn:
- how to build a synthesizer in a semester also a topic for everyone (PL students and others).

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
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

int[16] trans_sse(int[16] M) {
    int[16] S = 0, T = 0;
    S₀ = wr4(S , shufps(rd4(M, mx1₀), rd4(M, mx2₀), mi₀), 0);
    S₁ = wr4(S₀, shufps(rd4(M, mx1₁), rd4(M, mx2₁), mi₁), 4);
    S₂ = wr4(S₁, shufps(rd4(M, mx1₂), rd4(M, mx2₂), mi₂), 8);
    S₃ = wr4(S₂, shufps(rd4(M, mx1₃), rd4(M, mx2₃), mi₃), 12);
    T₀ = wr4(T , shufps(rd4(S₃, sx1₀), rd4(S₃, sx2₀), si₀), 0);
    T₁ = wr4(T₀, shufps(rd4(S₃, sx1₁), rd4(S₃, sx2₁), si₁), 4);
    T₂ = wr4(T₁, shufps(rd4(S₃, sx1₂), rd4(S₃, sx2₂), si₂), 8);
    T₃ = wr4(T₂, shufps(rd4(S₃, sx1₃), rd4(S₃, sx2₃), si₃), 12);
    assert equals(T₃, trans(M));
    return T₃;
}
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), si₂), 8);
    T₃ = wr4(T₂, shufps(rd4(S₃, sx1_3), rd4(S₃, sx2_3), si₃), 12);
    assert equals(T₃, trans(M));
    return T₃;
}

Encode M, trans(M), S₀ and T₀ as arrays variables with integer indices and value.
Encode mx1, mx2, mi, sx1, sx2 and si holes as integer variables.
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.
The name of the course this CS294 topics course has been listed as CS294: Programming Language Design for Everyone.

Since putting the course on the books, we are ready to teach a superset of intended material. In addition to designing domain-specific languages (DSLs) and their lightweight implementation, you will learn how to build a synthesizer in a semester, also a topic for everyone (PL students and others).

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?
The name of the course this CS294 topics course has been listed as CS294: Programming Language Design for Everyone. Since putting the course on the books, we realized we are ready to teach a superset of intended material.

In addition to:
- design of domain-specific languages (DSLs) and
- their lightweight implementation,

you will learn:
- how to build a synthesizer in a semester.

Also, a topic for everyone (PL students and others):
- 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)))

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

; 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.
The name of the course this CS294 topics course has been listed as CS294: Programming Language Design for Everyone. Since putting the course on the books, we realized we are ready to teach a superset of intended material. In addition to:

- design of domain-specific languages (DSLs) and their lightweight implementation,
- you will learn: how to build a synthesizer in a semester also a topic for everyone (PL students and others).

From SIMD transpose to Z3 integers & arrays

```racket
#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)))
...

\(\text{(Array I V)}\) introduces an array sort with indices of sort I and values of sort V.

\(\text{(select a i)}\) 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 (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))

...
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)
(define-sort BV2 () (\_ BitVec 2))
(define-sort BV4 () (\_ BitVec 4))

; an mx1_j hole is a 4-bit value in [0..12]
(define-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]
(define-const mi_0_0 BV2)

\texttt{define-sort} introduces a name for a sort; (\texttt{\_ BitVec k}) is the sort of bitvectors of length k.
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))

; an mx1_j hole is a 4-bit value in [0..12]
(define-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]
(define-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))

; an \texttt{mx1\_j} hole is a 4-bit value in \([0..12]\)
(declare-const \texttt{mx1\_0} BV4)
(assert (and (bvule (_ bv0 4) \texttt{mx1\_0})
            (bvule \texttt{mx1\_0} (_ bv12 4))))

; an \texttt{mi\_j} hole holds four 2-bit values in \([0..3]\)
(declare-const \texttt{mi\_0\_0} BV2)

\textbf{Bitvectors are unsigned, so there is no need to assert that \texttt{mi\_0\_0} is in \([0 .. 3]\).}

\textbf{Why use bitvectors?}

\texttt{define-sort} introduces a name for a sort; (_ BitVec k) is the sort of bitvectors of length k.

(_ bvV n) returns a bitvector value V of length n
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)

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

(_ bvV n) returns a bitvector value V of length n

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
- SMT-COMP (find the best solver for your problem)

Overview of SMT terminology and approaches

SMT solvers
- Solvers used in the SIMD transpose experiments: Boolector, CVC3, STP, Z3.
A selection of Racket tutorials, tools, and documentation

- Matthew Flatt, *Quick: An Introduction to Racket with Pictures*.
- Matthew Flatt and PLT. *The Racket Reference*.
- *Profile: Statistical Profiler*.

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

- Danny Yoo. *F*ding up a Racket.
The name of the course this CS294 topics course has been listed as CS294: Programming Language Design for Everyone. Since putting the course on the books, we are ready to teach a superset of intended material. In addition to:

- design of domain-specific languages (DSLs) and their lightweight implementation

you will learn:

- how to build a synthesizer in a semester

also a topic for everyone (PL students and others)

References: program encodings (a tiny sample)

Bounded model checking of sequential programs using SAT/SMT


Bounded model checking of concurrent programs using SAT/SMT


SMT-based verification (no bounding)

- K. Rustan M. Leino, Peter Müller, and Jan Smans. Verification of Concurrent Programs with Chalice. FOSAD 2009.