Generating Size-Parameterized Functions for Circuit Simulation Using Template Haskell

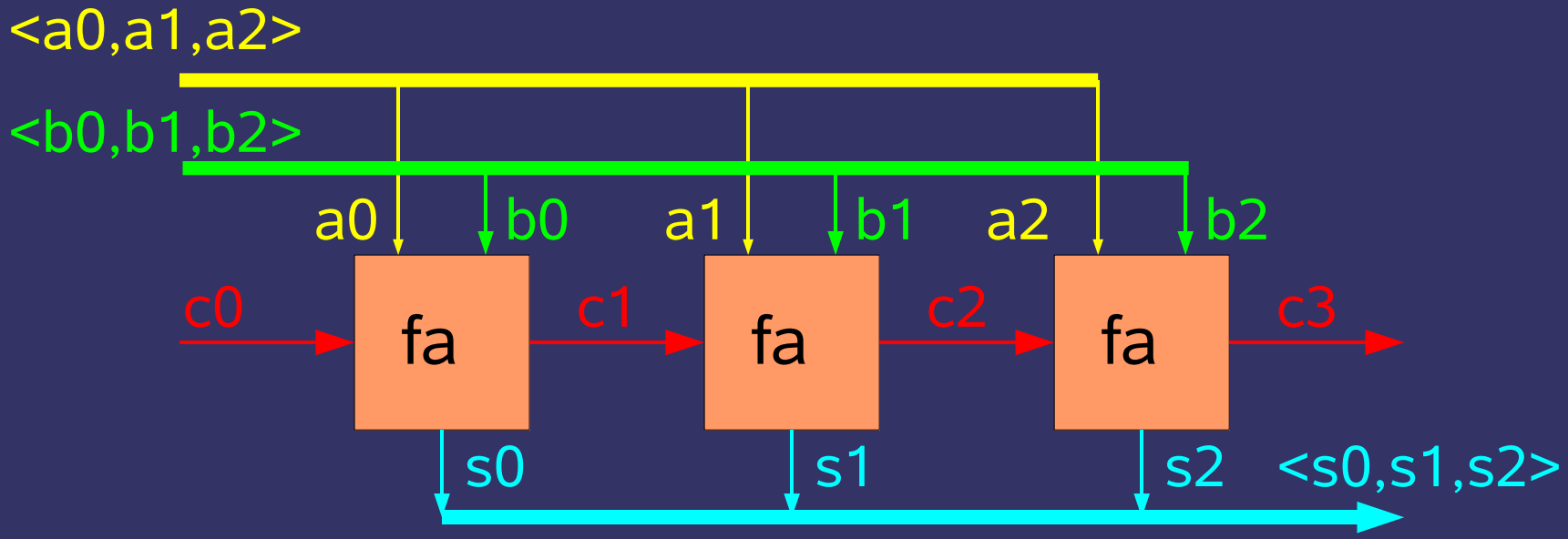
Christoph Herrmann
University of Passau

Talk at the IFIP-WG 2.11 Meeting
Dagstuhl, January 2006

Overview

- Problem
- Template Haskell
- Big Picture
- Main Skeletons
- Example Circuit Designs
- Index Transformations
- Conclusions

Problem: Representation of Wire Bundles



Choices: Lists, Arrays, Tuples

Pros and Cons of Bundle Representations

	List	Array	Tuple
flexible length	✓	✓	✗
type-safe length	✗	✗	✓
inhomogeneous	✗	✗	✓
efficiency	✗	?	✓

→ **use Template Haskell to make tuples flexible!**

Overview

- Problem
- Template Haskell
- Big Picture
- Main Skeletons
- Example Circuit Designs
- Index Transformations
- Conclusions

Template Haskell, Properties

- Homogeneous, two-level
- Manually annotated
- Static (compile-time) generation
- Lexical scoping
  - automatic renaming of local variables
  - cross-stage persistence
- Type safety of generated program
- Observability

Template Haskell, High-Level Annotations

- Quasi-Quotation
  - encloses part of object program
  - syntax: `[| \x -> x*x |]`
  - typechecked after construction of quoted part is complete
- Splicing
  - inserts code stored in meta variable in object part
  - syntax: `[| \dynarg -> $(fcode statarg) dynarg |]`
  - evaluated when code for quasi-quotation is constructed

Template Haskell, Simple Low-Level Example

Metaprogram

```
sel :: Int -> Int -> Q Exp
sel i n =
 do
  let as  = [ mkName ("a" ++ show j)
            | j<-[1..n] ]
      pat = TupP (map VarP as)
  return (LamE [pat] (VarE (as!!(i-1))))
```

Use in Application

```
... + $(sel 5 6) x - ...
```

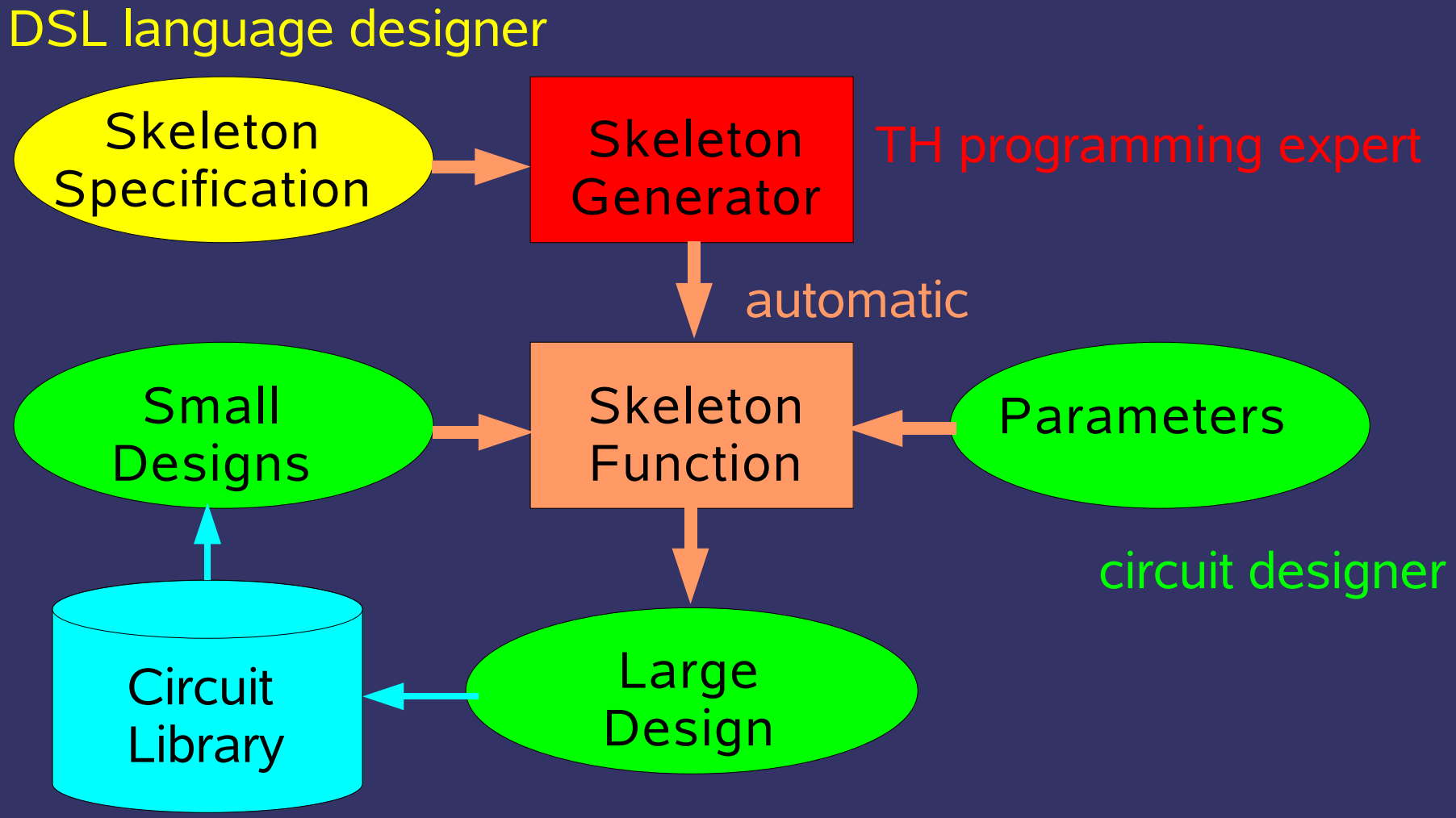
Generated Function

```
(\(a1, a2, a3, a4, a5, a6) -> a5)
```

Overview

- Problem
- Template Haskell
- Big Picture
- Main Skeletons
- Example Circuit Designs
- Index Transformations
- Conclusions

Big Picture


DSL language designer
Skeleton Specification
Skeleton Generator
TH programming expert
automatic
Small Designs
Skeleton Function
Parameters
circuit designer
Circuit Library
Large Design

Overview

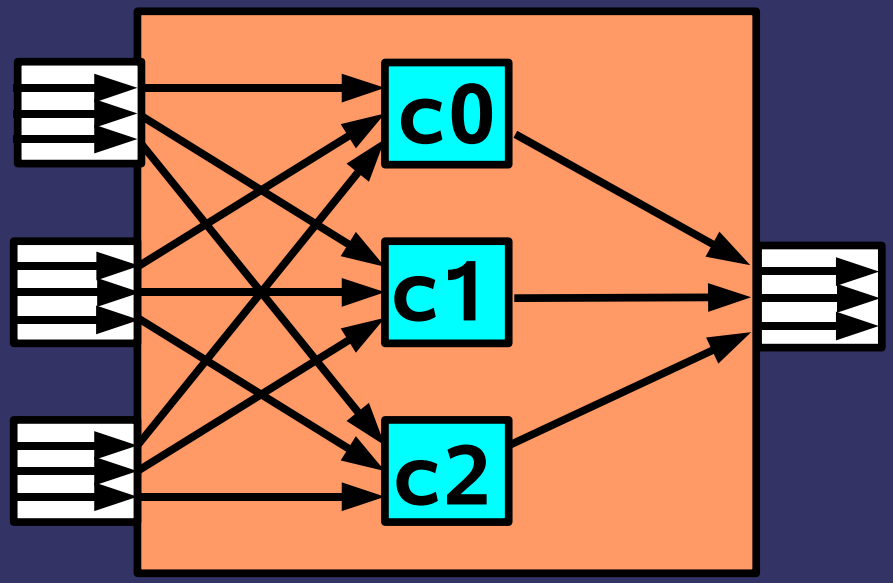
- Problem
- Template Haskell
- Big Picture
- Main Skeletons
- Example Circuit Designs
- Index Transformations
- Conclusions

Main Skeletons

- Design Composition (parallel,sequential)
  - parameters: degree, arity, component design
  - parallel: components are independent
  - sequential: components are connected linearly
  - wires are connected to bundles (tuples)
- Wiring Patterns
  - given: 2D-index transformation function (Haskell)
  - output: mapping between bundles of wire bundles

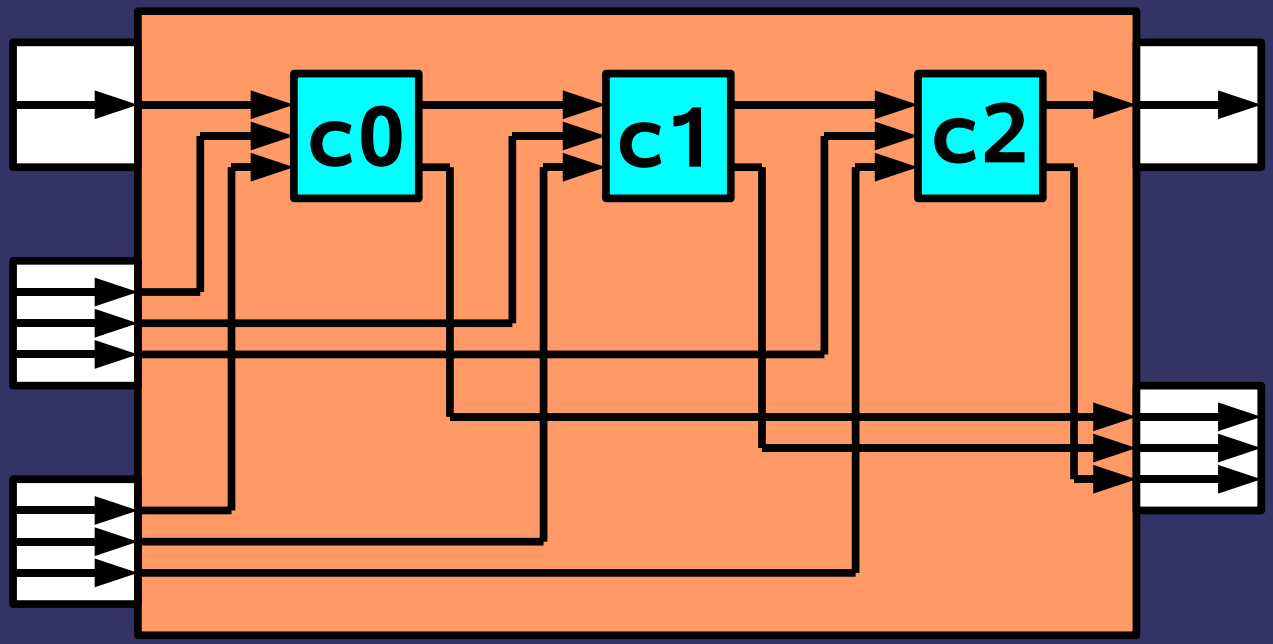
Parallel Composition Example

- parameters: degree=3, arity=3, component=`(\i-> c i)`
- corresponding ports are bundled in the composition
- component outports can be bundles themselves



Sequential Composition Example

- parameters: degree=3, arity=2, component=`(\i-> c i)`
- components are connected by their first ports
- other ports are bundled elementwise



Overview

- Problem
- Template Haskell
- Big Picture
- Main Skeletons
- Example Circuit Designs
- Index Transformations
- Conclusions

Example Design (1): n-bit Increment and Adder

```
wordInc, wordAdder :: Int -> Q Exp
wordInc    n =
  mSeq n 1 (const (\ [[c,x  ]] ->
                   [| halfAdder($c,$x   ) |]))
wordAdder n =
  mSeq n 2 (const (\ [[c,x,y]] ->
                   [| fullAdder($c,$x,$y) |]))
```

Example Design (1b): Splice

```
wordAdder n =
  mSeq n 2 (const (\ [[c,x,y]] ->
                   [| fullAdder($c,$x,$y) |]))


 $(wordAdder 3) ======>
  \ ~(a0,(x0_0, x0_1, x0_2),(x1_0, x1_1, x1_2))
  -> let
        ~(a1, y0) = fullAdder (a0, x0_0, x1_0)
        ~(a2, y1) = fullAdder (a1, x0_1, x1_1)
        ~(a3, y2) = fullAdder (a2, x0_2, x1_2)
     in (a3, (y0, y1, y2))
```

Example Design (2): n-bit Multiplexer

```
multiplex :: Int -> Q Exp
multiplex n =
  [| \ (sel,xs,ys)
      -> $(skelPar n 2 ( \_ [[x,y]] ->
                         [| mux(sel,$x,$y) |]))
         (xs,ys) |]
```

```
$(multiplex 3)  ======>
  \ (sel,xs,ys)
        -> \ ~((x0,x1,x2),(x3,x4,x5))
                -> let
                     ~x6 = mux (sel,x0,x3)
                     ~x7 = mux (sel,x1,x4)
                     ~x8 = mux (sel,x2,x5)
                   in (x6,x7,x8)
             (xs,ys)
```

Example Design (3): m×n-bit Sequential Multiplier

```
seqMultiplier m n =
 [| \ (zero,one,init,step,operand1,operand2)
    -> let (xLSB,_)  = $(shRegister m) (init,step,zero,$(wRev m) operand1)
           yReg      = $(register n) (init,operand2)
           maskedY   = $(skelPar n 1 (\_ [[yi]] -> [| and2(xLSB,$yi) |])) yReg
           newZ      = $(skelPar (m+n) 1 (\ _ [[zi]] ->
                                          [| and2(not1(init),$zi) |])) tmpRes
           zReg      = $(register (m+n)) (step,newZ)
           (zL,zH)   = $(wSplit m n) zReg
           (cy,sumN) = $(wordAdder n) (zero,maskedY,zH)
           tmpRes    = $(skelWire [m-1,n,1] (\ [_:l,h,c]->[l++h++c]))
                        (zL,sumN,cy)
           initCount = $(skelWire [2] (\[[z,e]] -> [(z:replicate (m-1) e)]))
                        (one,zero)
           (ready,_) = $(shRegister m) (init,step,zero,initCount)
           resultReg = $(register (m+n)) (ready,tmpRes)
       in (flipflop(step,ready),resultReg)
```

Overview

- Problem
- Template Haskell
- Big Picture
- Main Skeletons
- Example Circuit Designs
- Index Transformations
- Conclusions

Index Transformations (1)

- Specification: composition of wiring Functions
- Implementation: single function
- Solution
  - express wiring functions as `[[Int]]->[[Int]]`
  - compose them by Haskell composition
  - calculate their effect on static indices
  - generate function which establishes this effect

Index Transformations (2), Examples

- Split an (m+n) bundle into parts of sizes m and n

```
(zL,zH)   = $(wSplit m n) zReg
```

- Shift function on bundles of sizes (m-1), n, and 1

```
tmpRes    = $(skelWire [m-1,n,1]
                       (\ [_:l,h,c]->[l++h++c]))
              (zL,sumN,cy)
```

Overview

- Problem
- Template Haskell
- Big Picture
- Main Skeletons
- Example Circuit Designs
- Index Transformations
- Conclusions

Conclusions, from the User's Point of View

- Available
  - optional tool, compatible with hand-written designs
  - features by parameterization
    - 32 or 64 bit wordsize?
    - how many registers?
    - multiplier: how many bits in parallel?
- Still to be done:
  - domain-specific syntax and error messages
  - netlist generation

Conclusions, from the Implementer's Perspective

- Benefits from the expressive comfort in Haskell
  - compositionality (ex. wiring functions)
  - lexical scoping (ex. multiplexer)
- Similar to dependent types, without loss of
  - decidability of type inference
  - programming comfort (restrictions are local, imposed by skeletons)
- Parameterization without
  - need for extreme mathematical efforts
  - large case distinctions in the code
- Yet another example for the benefit of skeletons, especially if they are generated

Thank You for Your Attention!

Project page

http://www.infosun.fmi.uni-passau.de/cl/metaprog/