Supercompiling Haskell to Hardware

Arjan Boeijink, Philip K.F. Hölzenspies,Christiaan Baaij, Jan Kuper

University of Twente
Enschede

Implementation and Application of Functional Languages, 2013

Functional languages and hardware design

- Long history of functional hardware description languages
- To name but a few: Hawk, Lava, Kansas Lava, Wired and SAFL.
- "Hardware Design and FP: a Perfect Match" by Sheeran
- However each HDL has to make big tradeoffs in its implementation.

Different ways of combining hardware and FP

1. Executable specification only (Hawk)
   - Simple and expressive, but not synthesizable to hardware
2. Restricted versions of other languages (SAFL, Cλash)
   - Synthesizable, but limited in allowed abstractions
3. Embedded structural DSLs (Lava and variants, Wired)
   - Uses FP to generate hardware and no direct description
   - Synthesizable, but not pretty due to Haskell's lack of embeddings for pattern matching, guards, etc.
4. And more FP related ones (Ruby, Bluespec, reFlect, . . .)

Motivation

Creating the ideal hardware description language for myself

- Being able to use all host language features/abstractions
- Going from Haskell to hardware in minimal steps
- Anything that simulates should be synthesizable
- Full control over the behaviour of produced hardware
- From instruction set evaluator to a processor in a smooth process

Trying out supercompilation

- Searching for essence of translating FP to a hardware netlist
- Partial evalution is a good start, but seems not powerful enough
- Working with stream based abstractions requires perfect fusion
- Making supercompilation work in practice with an easier problem

What is supercompilation?

- A very general whole program optimization technique
- Superset of partial evaluation, deforestation, fusion, inlining, and more
- Basic ideas from Turchin in the 70s
- Recent work: "Supercompilation by Evaluation" by Max Bolingbroke

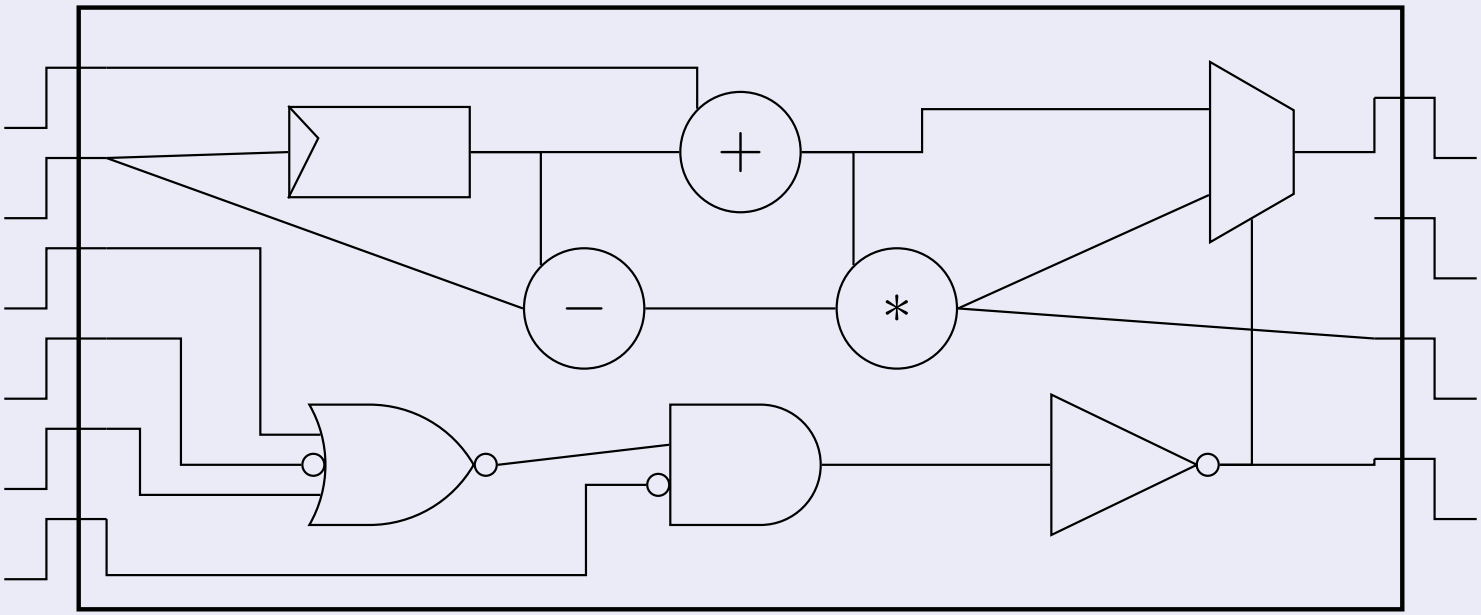
Informal descriptions

- Evaluating as much as possible of the program at compile time
- Very general transformation that removes abstraction (overhead)

Why is it not a widely used technique?

- Often super slow and super sized output code
- Hardly feasible on real programs
- Performance improvements vary a lot
- Termination concerns and preserving sharing makes it a hard problem

What is hardware?



Hardware properties

- Box with a graph of basic components
- Hardware size is bounded and determines the cost
- Fixed number of input pins and output pins
- All wires have monomorphic types and fixed width
- Netlist is a list of lowlevel components and its connections

Supercompilation and hardware?

Unconstrained supercompilation is translation to hardware

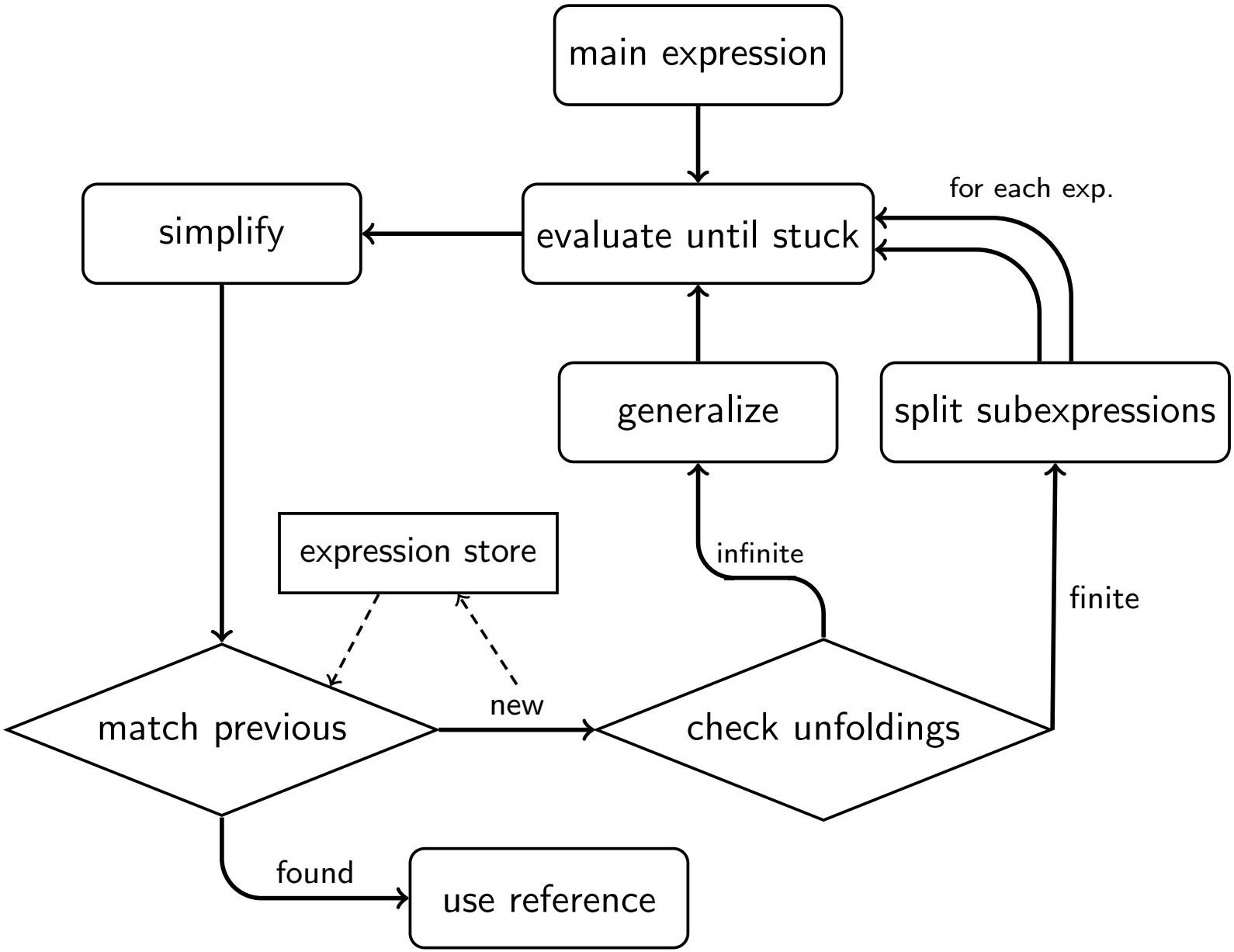
- A netlist is a finite program without any abstraction
- Supercompilation attempts to remove abstraction
- All abstraction can be removed by making SC unconstrained
- Guaranteed optimization can be used as translation

- We assume the source code makes sense as a hardware component
- And the code executes without error for all possible inputs

Hardware output simplifies supercompilation

- Hardware is finite so termination checks can be removed
- No code size explosion because hardware result is bounded
- Sharing of computation does not need to be preserved
  (because sharing can be easily recovered in the netlist)

Overview of the adapted supercompilation algorithm



Evaluating within a supercompiler

Evaluation state as central data structure

evaluation state: $\langle H \mid E \mid K \rangle$

H: heap (mapping from names to expressions)

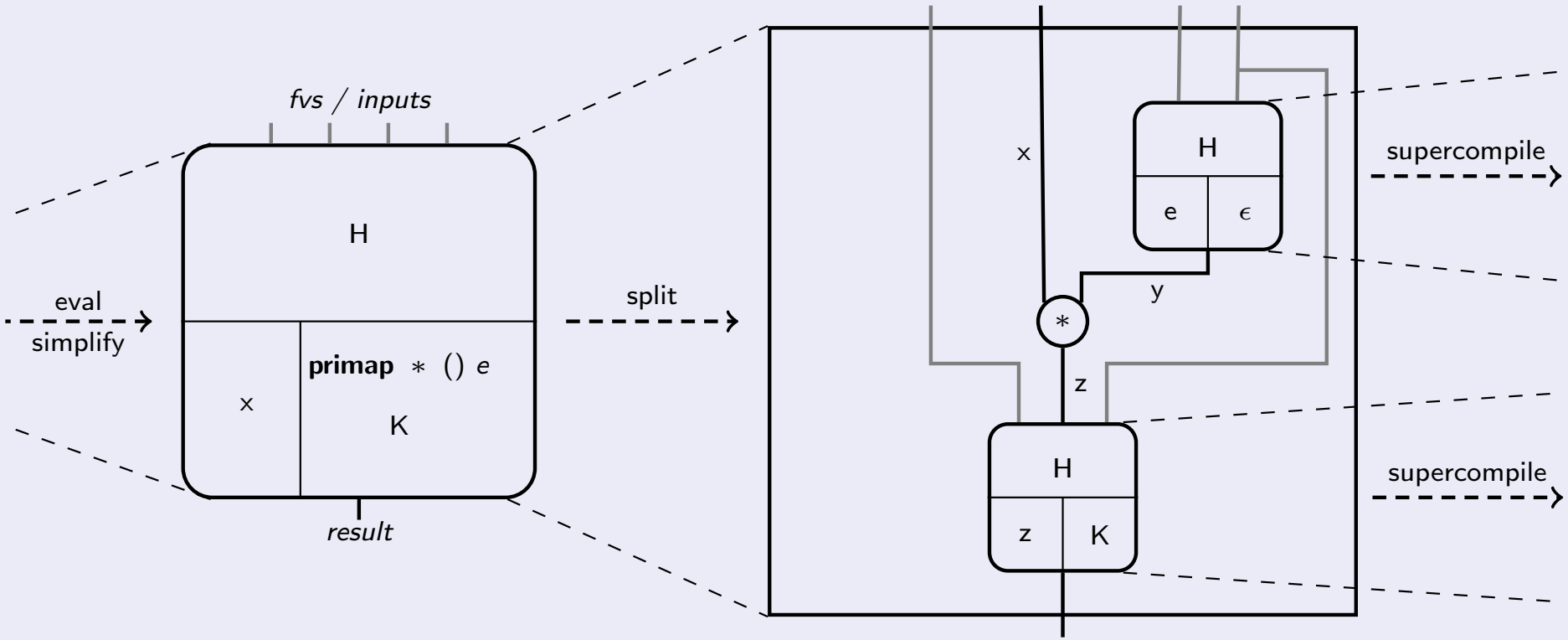
E: expression (the current expression in focus)

K: stack (list of frames $\approx$ rest of computation)

Operational semantics of the core language

VAR	$\langle H, x \overset{t}{\mapsto} e \mid x \mid K \rangle \rightsquigarrow \langle H \mid e \mid \mathbf{update}^t\ x, K \rangle$
UPDATE	$\langle H \mid v \mid \mathbf{update}^t\ x, K \rangle \rightsquigarrow \langle H, x \overset{t}{\mapsto} v \mid v \mid K \rangle$
APPLY	$\langle H \mid e\ x \mid K \rangle \rightsquigarrow \langle H \mid e \mid \mathbf{apply}\ x, K \rangle$
LAMBDA	$\langle H \mid \lambda x.e \mid \mathbf{apply}\ x, K \rangle \rightsquigarrow \langle H \mid e \mid K \rangle$
PRIM	$\langle H \mid \otimes\ (e_0, \overline{e}) \mid K \rangle \rightsquigarrow \langle H \mid e_0 \mid \mathbf{primapply}\ \otimes\ ()\ (\overline{e}), K \rangle$
PRIMMORE	$\langle H \mid \ell_n \mid \mathbf{primapply}\ \otimes\ (\overline{\ell})\ (e_n, \overline{e}), K \rangle \rightsquigarrow \langle H \mid e_n \mid \mathbf{primapply}\ \otimes\ (\overline{\ell}, \ell_n)\ (\overline{e}), K \rangle$
PRIMLAST	$\langle H \mid \ell_n \mid \mathbf{primapply}\ \otimes\ (\overline{\ell})\ (), K \rangle \rightsquigarrow \langle H \mid \mathrm{run}\ \otimes\ (\overline{\ell}, \ell_n) \mid K \rangle$
CASE	$\langle H \mid \mathbf{case}\ e\ \overline{a} \mid K \rangle \rightsquigarrow \langle H \mid e \mid \mathbf{scrutinise}\ \overline{a}, K \rangle$
DATA	$\langle H \mid C\ \overline{x} \mid \mathbf{scrutinise}\ \{\ldots, C\ \overline{x} \to e, \ldots\}, K \rangle \rightsquigarrow \langle H \mid e \mid K \rangle$
LITERAL	$\langle H \mid n \mid \mathbf{scrutinise}\ \{\ldots, n \to e, \ldots\}, K \rangle \rightsquigarrow \langle H \mid e \mid K \rangle$
LETREC	$\langle H \mid \mathbf{let}\ \overline{x =^t e}\ \mathbf{in}\ e' \mid K \rangle \rightsquigarrow \langle H, \overline{x \overset{t}{\mapsto} e} \mid e' \mid K \rangle$

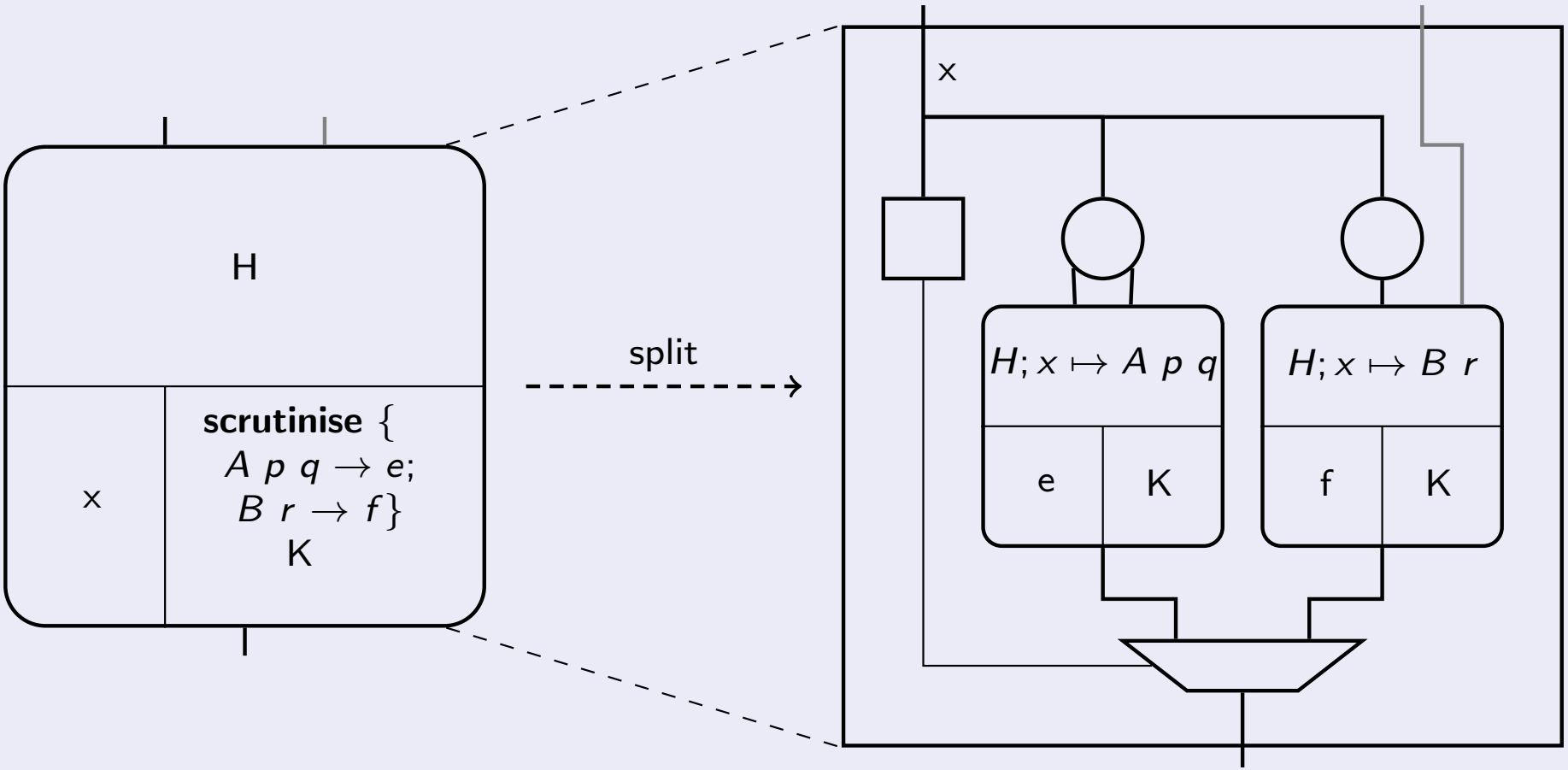
Intuition of Hardware Supercompilation



- Evaluate the source hardware description until you get stuck
- Stuck must mean that you are at input dependent expression
- Split the evaluation state in hardware components and substates
- Continue supercompiling the substates to produce hardware for them

Splitting case expressions

$$\frac{\langle H \mid x \mid \textbf{scrutinise}\ \{A\ p\ q \to e; B\ r \to f\}, K\rangle}{\begin{array}{l}\textbf{select}\ x\ \textbf{with}\\ \quad A\ p\ q \rightarrowtail \langle\!\langle x \mapsto A\ p\ q, H \mid e \mid K\rangle\!\rangle\\ \quad B\ r \rightarrowtail \langle\!\langle x \mapsto B\ r, \quad H \mid f \mid K\rangle\!\rangle\end{array}}$$



Dealing with some loss of sharing

- Duplication caused by cloning the heap for subcomponents
- The stack is copied when splitting case expressions

Duplication is not a critical problem

- Matcher reduces code size by reusing identical components during SC
- For the stack the duplication is limited by logic depth

Undoing the duplication

- Hardware SC produce a lot simple but nested components
- Simply flattening this structure could blowup exponentially
- Use bottom up transformation to merge identical components
- Identical components easy to detect due to matching in SC
- Also pull common components out of case expressions

Translating data types

User defined bit encoding for datatypes

```
class BitRepresentable a where
  type BitSize a :: Nat
  encode     :: a -> BitVector (BitSize a)
  decode     :: BitVector (BitSize a) -> a
```

Eliminating data constructor and case expressions

$$Foo\ a\ b \implies encode\ (Foo\ (decode\ a)\ (decode\ b))$$

$$\begin{array}{lcl} \textbf{select}\ x\ \textbf{with} & & \textbf{case}\ x\ \textbf{of} \\ Foo\ a\ b \rightarrowtail e & \implies & Foo\ a'\ b' \rightarrow \textbf{let}\ a = encode\ a' \\ & & \qquad\qquad\ \ \textbf{let}\ b = encode\ b' \\ Bar\ c \rightarrowtail f & & \qquad\qquad\ \ \textbf{in}\ e \\ & & Bar\ c' \rightarrow \textbf{let}\ c = encode\ c'\ \textbf{in}\ f \end{array}$$

Final step is just supercompiling it again and away

Streams as abstraction for synchronous signals

```
infixr 4 :<
data Signal a = a :< Signal a
instance Functor Signal where
  fmap f (x :< xs) = f x :< fmap f xs
instance Applicative Signal where
  pure x = x :< pure x
  (f :< fs) <*> (y :< ys) = f y :< (fs<*>ys)
```

Stream functions with state arguments

```
foo = f initX initY where
  f x y (a:<as) (b:<bs) =(c:<cs)
    where x' = updX a b x
          y' = updY b y
          c  = bar a x y
          cs = f x' y' as bs
```

Implicit stream result

foo a b $\leftrightharpoons$

bar a b x y **using**

$x \leftarrow \mathrm{delay}\ initX\ x'$

$x' \leftarrow updX\ a\ b\ x$

$y \leftarrow \mathrm{delay}\ initY\ y'$

$y' \leftarrow updY\ b\ y$

Rollback/generalization on accumulating state

```
acummulator xs = accum 0 xs where
  accum a (b :< bs) = let y = a + b in
    y :< accum y bs
```

Makes the supercompiler go through a series of evaluation states:

$$S^1: \langle a_0 \overset{t3}{\mapsto} 0,\ y_0 \overset{t5}{\mapsto} a_0 + b_0, \quad ys_0 \overset{t6}{\mapsto} accum\ y_0\ bs_0 \mid b_0 :< bs_0 \leftarrow xs_0 \mid y_0 :< ys_0 \mid \epsilon \rangle$$

$$S^2: \langle a_0 \overset{t3}{\mapsto} 0,\ y_0 \overset{t5}{\mapsto} a_0 + b_0, y_1 \overset{t5}{\mapsto} y_0 + b_1, \quad ys_1 \overset{t6}{\mapsto} accum\ y_1\ bs_1 \mid b_1 :< bs_1 \leftarrow bs_0 \mid y_1 :< ys_1 \mid \epsilon \rangle$$

$$S^3: \langle a_0 \overset{t3}{\mapsto} 0,\ y_0 \overset{t5}{\mapsto} a_0 + b_0, y_1 \overset{t5}{\mapsto} y_0 + b_1, y_2 \overset{t5}{\mapsto} y_1 + b2, ys_2 \overset{t6}{\mapsto} accum\ y_2\ bs_2$$

...

Rolling back to avoid this nontermination in the SC process

- Gathering tagbags: S^1: $\{1 * t3, 1 * t5, 1 * t6\}$ S^2: $\{1 * t3, 2 * t5, 1 * t6\}$
- Growing tags and being present across cycles, assigns blame to y_0
- Rolling back and split off y_0 from S^1, then resume supercompilation

Avoiding initial state propagation

```
avgLast4 :: Signal Int -> Signal Int
avgLast4 = avg4 0 0 0 where
  avg4 a b c (x :< xs) =
    div (x+a+b+c) 4 :< avg4 x a b xs
```

This results in the recursion being unpeeled three times:

$$
\begin{array}{lll}
avgLast4 & (x :< xs) = div(x+0+0+0)4 :< f2\ x & xs \\
f2\ a & (x :< xs) = div(x+a+0+0)4 :< f3\ x\ a & xs \\
f3\ a\ b & (x :< xs) = div(x+a+b+0)4 :< f4\ x\ a\ b & xs \\
f4\ a\ b\ c & (x :< xs) = div(x+a+b+c)4 :< f4\ x\ a\ b & xs
\end{array}
$$

Solving this needless component duplication

- This kind of duplication is common and very bad for hardware area
- On finding the recursion, search for earlier specialized states
- Here rolling back to the first state and splitting off the zeros
- Both kinds of rollback use an iterative process to reach final form

Disadvantages of hardware supercompilation

Naive code can produce massive lookup tree

```
bitCount :: Int32 -> Int32
bitCount x = bc x nat32 where
  bc _ Z     = 0
  bc a (S c) = (if even a then 0 else 1) +
    bc (a `shiftR` 1) c
```

This example produces a binary tree with 'even a' on the nodes and a concrete number on the leaves (all additions are gone)

- Branching in a finite recursion seems easy to avoid in practice

Supercompilation is the sledgehammer approach

- Guaranteed success but do not expect pretty results
- All structure is lost because SC uses only the semantics
- Requires some experience to use safely

Any high level language to hardware

Unconstrained supercompilation could translate any language to hardware
Nothing of this technique is Haskell specific

Language requirements:

- Operational semantics for the input language (simple is better)
- An output language you can synthesize hardware from (preferably close to a subset of the input language)

Things to implement:

- Evaluator for the input language (just implement the semantics)
- Matcher for the eval-state (on state structure, modulo naming)
- Optionally a simplifier (standard things like GC and DCE)
- The splitter (tricky, but follows structure of eval-state and output)
- Basic optimizer for the output (if you care about area)

Conclusions

Proposed new method to produce hardware from Haskell

- No embedding allows for more expressive hardware descriptions
- Simple principled method determined only by the semantics
- Supercompilation can be used as translation step
- Adapted supercompilation for another practical application
- Not the ideal method, but an alternative with new tradeoffs

Perfect triple?: Hardware design, functional programming and SC

- Functional languages are good at generative abstractions
- Direct correspondence combinatorial and functional
- Supercompilation is easier with simple semantics
- Hardware domain avoids supercompilation disadvantages
- Constructing netlists $\approx$ removing all abstraction $\approx$ unconstrained SC

Future work

Pratical improvements

- Support more language constructs (arrays, mutable variables)
- High level optimizations before going to synthesis
- More abstractions and EDSLs on top of this
- Use better type level numerals
- Warnings on potential infinite hardware
- Dependently typed language instead?

Further research on supercompilation

- Apply lessons learned to general purpose SC
- Proving properties of this SC variant
- Selective supercompilation over other EDSLs
- Making the supercompilation process faster

Thank you

Questions?