Supercompiling Haskell to Hardware

Arjan Boeijink Philip K.F. Hölzenspies Christiaan Baaij Jan Kuper
University of Twente
{w.a.boeijink,p.k.f.holzenspies,c.p.r.baaij,j.kuper}@utwente.nl

Abstract

Supercompilation is a global optimization technique which removes abstractions from a program. A hardware design can be viewed as a first order program without any abstraction. Hence, in principle, unconstrained supercompilation is able to generate hardware from a total Haskell program. The properties that hold in the hardware domain make it possible for us to sidestep the practical complications, that supercompilation has for general programs. We present a version of supercompilation which transforms Lava-style hardware descriptions – with signal types exposed as streams – into hardware, without being restricted by the limitations that follow from deeply embedding a domain specific language in Haskell. We show that a modified supercompilation algorithm can be used as the main step in translating Haskell to hardware. Supercompilation enables using more of Haskell to describe hardware, including recursion, and choice constructs such as pattern matching and guards.

In short, hardware makes supercompilation easier and supercompilation allows synthesis of more elegant hardware descriptions.

Categories and Subject Descriptors B.6.3 [*Logic Design*]: Design Aids—Hardware description languages; D.3.2 [*Programming Languages*]: Language Classifications—Applicative(functional) languages

General Terms Supercompilation, Hardware design

1. Introduction

The history of functional Hardware Description Languages (HDLs) is long and has produced many tools and languages. To name but a few: Hawk, Lava, Kansas Lava, Wired and SAFL[1, 3, 10]. Hardware and functional languages match very well with each other, however each HDL had to make big tradeoffs in its implementation. These languages can roughly be categorised in three groups:

1. Specification only (elegant and expressive, but not synthesizable)
2. Restricted versions of other languages (synthesizable, but lacking abstraction)
3. Embedded structural DSLs (synthesizable, but often onerous to use, especially due to the lack of deep embeddings for powerful language constructs, such as pattern matching, guards, etc.)

The context for the work presented in this paper is CλaSH[2], but it applies equally to Lava-style HDLs. CλaSH falls in the second category above, in that it is a restricted form of Haskell. It is used to describe synchronous hardware, by defining functions on *streams*. Streams are given the special interpretation of temporally spaced input, i.e. an input to the synchronous system under design is a stream, where the next element in the stream is the input on the next clock cycle. Functions may be defined on a set of input and output streams, but may not violate causality. In other words, they may neither 'lose' time, nor see into the future (they consume precisely one element from every input stream and produce precisely one element on every output stream).

'Between' streams, CλaSH is 'plain' Haskell, albeit with restrictions. Recursion is generally not allowed. Instead, the language provides specialised versions of common higher-order functions, like `map` and `fold`. The main reason to disallow recursion, is the lack of a sufficiently strong normalisation that guarantees recursive code is synthesizable.

In this paper, we present a new approach, that alleviates this—and similar—restrictions. The approach is based on the application of supercompilation as the normalisation of CλaSH programs. Acceptable programs must be total (between consecutive stream consumptions and/or productions). Under this restriction, however, we allow for any form of abstraction, e.g. general recursion, (list) comprehension, etc.

1.1 Motivation for using supercompilation

This work was initiated by a discussion on: what is the essence of translating a functional program to a hardware netlist? Partial evaluation of higher order arguments (and other things that can not exist in hardware) comes close but seems not powerful enough. Describing synchronous hardware using streams as signals requires complete fusion of all steam functions. Recent advances in supercompilation [11] and the fact that supercompilation covers more than partial evaluation and fusion, made it an interesting option. However supercompilation has been mostly used as an optimization (not a translation) technique, and is not yet practical for general use (because of code size and speed issues). With no strong arguments for why supercompilation could not be the solution for translation hardware, we had to try it.

The power of supercompilation of lazy functional language such as in "Supercompilation by Evaluation" [5] is bounded by the termination criteria and the preservation of sharing. By assuming that a sensible structural hardware description is a total program, we can ignore the restrictions imposed by the termination checker. Because the resulting netlist is (after flattening) a graph of basic components, we can recover the sharing after the supercompilation by unifying components with identical inputs. Without the limitations imposed by these concerns we can use supercompilation as a complete step in translating Haskell to hardware.

The next section starts with an introduction into supercompilation, which may be skipped if the reader is already familiar with

function definition:

$$d \quad ::= \quad f\ x^\star = e$$

expressions:

$$\begin{array}{lll} e & ::= & v & \textit{(value)} \\ & | & x & \textit{(variable)} \\ & | & \mathbf{let}\ \overline{x =^t e}\ \mathbf{in}\ e & \textit{(recursive let expr.)} \\ & | & \mathbf{case}\ e\ \mathbf{of}\ a^+ & \textit{(case expr.)} \\ & | & e\ x & \textit{(application)} \\ & | & \otimes\ e^+ & \textit{(primitive operation)} \\ & | & \pi_n\ e & \textit{(proj. of a product type)} \end{array}$$

case alternatives:

$$\begin{array}{lll} a & ::= & C\ x^\star \to e & \textit{(constructor alternative)} \\ & | & n \to e & \textit{(integer alternative)} \\ & | & \mathbf{default} \to e & \textit{(default alternative)} \end{array}$$

values:

$$\begin{array}{lll} v & ::= & \lambda x \to e & \textit{(lambda abstraction)} \\ & | & n & \textit{(integer)} \\ & | & C\ x^\star & \textit{(constructor (saturated))} \end{array}$$

Figure 1. Grammar of the core language

the recent works on supercompilation. Readers more interested in using hardware description languages or in how supercompilation allows for more elegance may want to look at the larger example in section 6.

2. Supercompilation

Supercompilation originates from the work of Turchin [16] in the 1970s and constitutes a very general whole-program optimization technique. It is a generalization of partial evaluation [9] and is a superset of many popular optimizations, like deforestation, fusion, inlining, constant propagation and specialization. It works by aggressively evaluating as much of a program as possible at compile-time. Where parts of the program are input-dependent, different cases are explored and compiled separately, later to be merged back into a single program. We would like to strongly encourage readers not familiar with supercompilation to read the recent paper "Supercompilation by Evaluation" [5] for a good explanation of the main concepts. The main reasons that supercompilation has not found universal uptake, is that it is typically prohibitively slow and often produces enormously inflated results. For real-world programs, it is by and large considered infeasible, especially because the gains (mostly in terms of improved performance) vary tremendously across programs that, on superficial inspection, seem highly similar. Under the constraints for hardware descriptions, however, these pitfalls are avoided; synthesizable hardware is inherently finite and has no higher-order abstractions.

2.1 Functional core language and evaluator

The input of our supercompiler is the simple functional core language shown in Figure 1. The core language is in applicative normal form and has an unique tag attached to each let binding. Figure 2 shows the operational semantic for this core language in the style of Sestoft [13]. The tags are preserved in heap binding and update frames.

2.2 Termination

There are two different sources of nontermination in supercompilation. The first is the evaluation of a term that is non-terminating. The second is nontermination in the splitter. Having good termination criteria is a crucial aspect of a supercompilation implementation. See 'Termination combinators forever'[7] for more on this topic.

2.3 Problems with supercompilation

The main problem with supercompilation is the code size explosion for many input programs. Like other whole-program optimization techniques, compilation times are often very long, but supercompilation can be prohibitively so. The benefits of supercompilation vary a lot between programs, many programs run only a few percent faster with significant code size increase. The optimization capabilities of supercompilation are limited by the necessarily conservative termination criteria. Other requirements like preservation of sharing, can inhibit a lot of potential optimizations. Another problem is that the output of supercompilation tends to be more difficult to optimize for lower level optimization passes.

3. Supercompilation to hardware

Synthesizable hardware is usually described by means of a *netlist*. Netlists are total programs without any abstraction, because they describe finite and concrete hardware. Consequently, the result of our supercompilation should be a netlist, or a program that has a straightforward transformation to netlists. To this end, we introduce NETCORE, which is a core language in *hardware normal form* presented in Figure 3.

The input of our supercompiler is similarly constrained by these properties of the desired result. All types used must have hardware representations. In other words, every type must have a bijection to bit vectors. This excludes the use of IO operations (whether hidden by unsafe operations or not), which is no limitation on the output, considering that all transistors are non-side-effecting. Since recursion in the output language would correspond to combinatorial loops in hardware, the input is limited to *total* programs only.

As an intuition for the correspondence between NETCORE and actual hardware, consider functions as components and application as instantiation. This means that functions in NETCORE must be *fully* applied and all types in NETCORE have fixed-size bit vector representations.

3.1 Intuition of hardware supercompilation

The supercompiler starts with evaluating the hardware description until it gets stuck. When it is stuck we want to be sure that it has arrived at an evaluation state which represents some hardware component depending on some unknown input. This means that the evaluator may not be blocked by a termination checker or lack of information. The splitter produces the hardware corresponding to this evaluation state with optionally subcomponents that are evaluation states still to supercompiled further. The splitter needs to be minimal in the hardware it peels of the evaluation state and pass as much information as possible to the substates. Figure 4 show a graphical and hardware oriented view of the intuition behind using supercompilation as translation.

3.2 The toplevel

Figure 5 shows the toplevel code of our supercompiler. Compared with algorithm presented in 'supercompilation by evaluation', the main difference are: no termination checking at all, evaluation is always done before matching and split is defined as a pure function.

3.3 Improving the matcher

Hardware designs tend to consist of many components, often with groups of components that are almost identical (for example differing only in some constants). This can make matching very slow, because the matcher attempts to match with every previous evaluation state, thus has quadratic behaviour in the number of components. While it is more common in the hardware domain, this quadratic matcher behaviour could be a problem for any supercompiler. The solution to this problem is generating a structural (ignoring names

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)

stack frame:

$$
\begin{array}{lcll}
k & ::= & \mathbf{update}^t\ x & \textit{(update frame)} \\
 & \mid & \mathbf{apply}\ x & \textit{(application frame)} \\
 & \mid & \mathbf{primapply}\ \otimes\ ()\ e & \textit{(apply left operand to primop)} \\
 & \mid & \mathbf{primapply}\ \otimes\ c\ () & \textit{(apply right operand to primop)} \\
 & \mid & \mathbf{scrutinise}\ a^+ & \textit{(scrutinise value)}
\end{array}
$$

$$
\begin{array}{ll}
\textsc{var} & \langle H, x \overset{t}{\mapsto} e \mid x \mid K \rangle \rightsquigarrow \langle H \mid e \mid \mathbf{update}^t\ x, K \rangle \\
\textsc{update} & \langle H \mid v \mid \mathbf{update}^t\ x, K \rangle \rightsquigarrow \langle H, x \overset{t}{\mapsto} v \mid v \mid K \rangle \\
\textsc{apply} & \langle H \mid e\ x \mid K \rangle \rightsquigarrow \langle H \mid e \mid \mathbf{apply}\ x, K \rangle \\
\textsc{lambda} & \langle H \mid \lambda x.e \mid \mathbf{apply}\ x, K \rangle \rightsquigarrow \langle H \mid e \mid K \rangle \\
\textsc{prim} & \langle H \mid \otimes\ (e_0, \overline{e}) \mid K \rangle \rightsquigarrow \langle H \mid e_0 \mid \mathbf{primapply}\ \otimes\ ()\ (\overline{e}), K \rangle \\
\textsc{primmore} & \langle H \mid \ell_n \mid \mathbf{primapply}\ \otimes\ (\overline{\ell})\ (e_n, \overline{e}), K \rangle \rightsquigarrow \langle H \mid e_n \mid \mathbf{primapply}\ \otimes\ (\overline{\ell}, \ell_n)\ (\overline{e}), K \rangle \\
\textsc{primlast} & \langle H \mid \ell_n \mid \mathbf{primapply}\ \otimes\ (\overline{\ell})\ (), K \rangle \rightsquigarrow \langle H \mid \mathsf{run}\ \otimes\ (\overline{\ell}, \ell_n) \mid K \rangle \\
\textsc{case} & \langle H \mid \mathbf{case}\ e\ \overline{a} \mid K \rangle \rightsquigarrow \langle H \mid e \mid \mathbf{scrutinise}\ \overline{a}, K \rangle \\
\textsc{data} & \langle H \mid C\ \overline{x} \mid \mathbf{scrutinise}\ \{\ldots, C\ \overline{x} \to e, \ldots\}, K \rangle \rightsquigarrow \langle H \mid e \mid K \rangle \\
\textsc{literal} & \langle H \mid n \mid \mathbf{scrutinise}\ \{\ldots, n \to e, \ldots\}, K \rangle \rightsquigarrow \langle H \mid e \mid K \rangle \\
\textsc{letrec} & \langle H \mid \mathbf{let}\ \overline{x =^t e}\ \mathbf{in}\ e' \mid K \rangle \rightsquigarrow \langle H, \overline{x \overset{t}{\mapsto} e} \mid e' \mid K \rangle
\end{array}
$$

Figure 2. Operational semantics of the core language

component definition:

$$c \quad ::= \quad f\ x^\star \ \leftrightharpoons\ b$$

component body:

$$
\begin{array}{lcll}
b & ::= & i & \textit{(component instantiation)} \\
 & \mid & i\ \mathbf{using}\ \overline{x \leftharpoondown i} & \textit{(component composition)} \\
 & \mid & \mathbf{select}\ x\ \mathbf{with}\ a^+ & \textit{(multiplexed components)}
\end{array}
$$

select alternative:

$$
\begin{array}{lcll}
a & ::= & e \rightarrowtail b & \textit{(enum alternative)} \\
 & \mid & \mathbf{default} \rightarrowtail b & \textit{(default alternative)}
\end{array}
$$

component instantiation:

$$
\begin{array}{lcll}
i & ::= & x & \textit{(variable)} \\
 & \mid & n & \textit{(numeric constant)} \\
 & \mid & f\ x^\star & \textit{(component instantiation)} \\
 & \mid & p\ x^+ & \textit{(primitive operator)} \\
 & \mid & C\ x^\star & \textit{(composition using constructor)} \\
 & \mid & x :< xs & \textit{(stream production)}
\end{array}
$$

extensions for intermediate NETCORE:

$$
\begin{array}{lcll}
b & ::= & \ldots & \\
 & \mid & \overline{x :< xs \leftarrow ys}\ \mathbf{in}\ b & \textit{(stream consumption)} \\
 & \mid & \lambda x.b & \textit{(lambda abstraction)} \\
a & ::= & \ldots & \\
 & \mid & C\ x^\star \rightarrowtail b & \textit{(constructor alternative)} \\
i & ::= & \ldots & \\
 & \mid & x :< xs & \textit{(stream production)} \\
 & \mid & C\ x^\star & \textit{(composition using constructor)}
\end{array}
$$

Figure 3. Grammar of the hardware normal form

and heap ordering) hash for each evaluation state in the history. Using this hash, the number of potential matches for an evaluation state is reduced to a few and often zero or one. We calculate this structure hash in two steps: first, for every heap binder, the expression is hashed and then, using the extra information from heap bound variables, the hash for the whole evaluation state is calculated. The multistep hash calculation is essential to take the structure of the heap into account, while ignoring names and the order of the heap binders.

3.4 Simplification of evaluation states

To improve the chances of matching evaluations states, various simplification steps are applied before matching.

Alias elimination Trivial heap bindings with only a variable as expression can be eliminated by renaming them in the whole evaluation state. A similar situation can happen on the stack with consecutive update frames, that can be merged into a single update frame by renaming. The third opportunity for renaming occurs when the term to be evaluated is a free variable and the top of the stack is an update frame.

Heap garbage collection The evaluator can leave dead bindings on the heap, and the splitter does not filter the heap binding for the substates. We use simple copying garbage collector to remove all unreachable binding from the heap.

Unused updates removal The stack can contain update frames for let bindings that turned out to be not shared. These update frames can be filtered out, by simple checking if the variable to be updated is not a free variable of the evaluation state.

3.5 The new splitter

The splitter is redesigned because it has to translate from the core language to hardware normal form. The second big difference with other splitters is that it may not residualize anything that could prevent optimizations of the subexpressions. Because ensuring future optimizations is crucial for the completeness of the translation process, the splitter can not preserve sharing. The input of the split function is an evaluation state that can not be reduced further by the rules from the operational semantics. This irreducibility of the input makes the splitter slightly simpler.

For the splitter cases we start with the input evaluation state, with optional side conditions. And below that the NETCORE result of the splitting, where $\langle\!\langle H \mid e \mid K \rangle\!\rangle$ is the component instantiation resulting from supercompiling the substate $\langle H \mid e \mid K \rangle$.

Trivial splittings Splitting of evaluation states with an empty stack and heap, results in only a residual expression and no substates. A state with a free variable in an empty context results in that variable. If the expression is an constant value, then the splitter just returns that constant as residual expression.

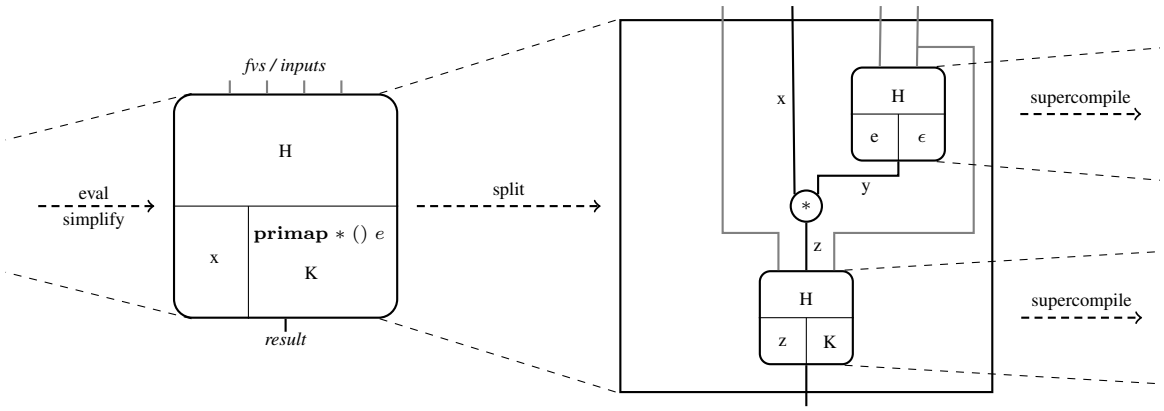

Figure 4. Graphical representation of hardware supercompilation process centered on the splitter

```
reduce :: EvalState -> EvalState
simplify :: EvalState -> EvalState
match :: EvalState -> EvalState -> Maybe Renaming
split :: EvalState -> Bracket

data Bracket = B {holes :: [EvalState],
    assemble :: [Instantiation] -> ResultExp}
data Promise = P {funName :: String,
    freeVars :: [Name], meaning :: EvalState}

sc :: EvalState -> SCM Instantiation
sc s = do
  let state = simplify (reduce s)
  ps <- gets promises
  case findMatchingState state ps of
    Just (p, rn) ->
      return (ComplInst (funName p)
        (map (rename rn) (freeVars p)))
    Nothing -> do
      let fvs = stateFVs state
      n <- promise fvs state
      let bracket = split state
      ys <- mapM sc (holes bracket)
      bind n fvs (assemble bracket ys)
      return (ComplInst n fvs)
```

Figure 5. Code for toplevel of the supercompiler

$$\underset{x}{\langle \epsilon \mid x \mid \epsilon \rangle} \qquad \underset{n}{\langle \epsilon \mid n \mid \epsilon \rangle}$$

Splitting values Splitting of values is relatively easy because the evaluation stack is empty. Constructor values are split by making a substate for each argument and the residual is the constructor with the outputs of the new components as arguments.

$$\underset{\begin{array}{l} C\ x\ y\ \textbf{using} \\ \quad x \leftarrow \langle\!\langle H \mid a \mid \epsilon \rangle\!\rangle \\ \quad y \leftarrow \langle\!\langle H \mid b \mid \epsilon \rangle\!\rangle \end{array}}{\langle H \mid C\ a\ b \mid \epsilon \rangle}$$

Lambdas are split with the body of the lambda as a substate and the residual is a component with the lambda variable as first input. Only top level defined functions may have lambdas as user defined components. Because states given to the splitter have been fully evaluated, all other lambdas have been beta reduced away.

$$\underset{\lambda\ x\ y\ .\ \langle\!\langle H \mid e \mid \epsilon \rangle\!\rangle}{\langle H \mid \lambda\ x\ y\ \rightarrow\ e \mid \epsilon \rangle}$$

Splitting applications If the top of the stack is an application frame then a chain of all consecutive application frames will be gathered. The result of the splitting is an substate for each argument in the chain, and the residual is an component instantiation.

$$\underset{\begin{array}{l} \langle\!\langle H \mid z \mid K \rangle\!\rangle\ \textbf{using} \\ \quad x \leftarrow \langle\!\langle H \mid a \mid \epsilon \rangle\!\rangle \\ \quad y \leftarrow \langle\!\langle H \mid b \mid \epsilon \rangle\!\rangle \\ \quad z \leftarrow f\ x\ y \end{array}}{\langle H \mid f \mid \textbf{apply}\ a, \textbf{apply}\ b, K \rangle}$$

One problem is that a chain of application frames can be interleaved by one or more update frames. These update frames can be removed by rebuilding the applications above the update frame as heap bindings.

$$\underset{\begin{array}{l} \langle\!\langle H \mid z \mid K \rangle\!\rangle\ \textbf{using} \\ \quad x \leftarrow \langle\!\langle H \mid a \mid \epsilon \rangle\!\rangle \\ \quad y \leftarrow \langle\!\langle w \overset{t}{\mapsto} f\ x, H \mid b \mid \epsilon \rangle\!\rangle \\ \quad z \leftarrow f\ x\ y \end{array}}{\langle H \mid f \mid \textbf{apply}\ a, \textbf{update}^t\ w, \textbf{apply}\ b, K \rangle}$$

An update frame at the end of an application chain is removed too, with the update variable (instead of a fresh variable) being bound to the residual application.

$$\underset{\begin{array}{l} \langle\!\langle H \mid w \mid K \rangle\!\rangle\ \textbf{using} \\ \quad x \leftarrow \langle\!\langle H \mid a \mid \epsilon \rangle\!\rangle \\ \quad y \leftarrow \langle\!\langle H \mid b \mid \epsilon \rangle\!\rangle \\ \quad w \leftarrow f\ x\ y \end{array}}{\langle H \mid f \mid \textbf{apply}\ a, \textbf{apply}\ b, \textbf{update}^t\ w, K \rangle}$$

Splitting primitive operations Primitive operations are split in a similar way to applications. In the first case the left operand is open and the second operand an expression.

$$\langle H \mid x \mid \mathbf{primapply}\ \otimes\ ()\ e, K\rangle$$
$$\langle\!\langle H \mid z \mid K\rangle\!\rangle\ \mathbf{using}\ y \leftarrow \langle\!\langle H \mid e \mid \epsilon\rangle\!\rangle,\ z \leftarrow x \otimes y$$

In the other case the first operand is an constant value and the second operand open.

$$\langle H \mid y \mid \mathbf{primapply}\ \otimes\ c\ (), K\rangle$$
$$\langle\!\langle H \mid z \mid K\rangle\!\rangle\ \mathbf{using}\ x \leftarrow c,\ z \leftarrow x \otimes y$$

Like with applications, if an update frame is next on the stack, it is taken off and its variable is used to bind the residual operator to.

$$\langle H \mid x \mid \mathbf{primapply}\ \otimes\ ()\ e, \mathbf{update}^t\ a, K\rangle$$
$$\langle\!\langle H \mid a \mid K\rangle\!\rangle\ \mathbf{using}\ y \leftarrow \langle\!\langle H \mid e \mid \epsilon\rangle\!\rangle,\ a \leftarrow x \otimes y$$

Recursive lets in splitting Update frames originating from recursive let bindings need special care to preserve correctness. These update frames can be referred to from the term being evaluated or another frame higher up the stack, both directly or indirectly through heap bindings. The stack is split at every recursive update frame, these substacks are residualized to subcomponents. Splitting the stack may cause update frames in a substack to be referred to from other substacks. Substacks need to be split again at such update frames.

$$\langle H \mid e \mid K_0, \mathbf{update}^t\ x, K_1, \mathbf{update}^u\ y, K_2\rangle \quad y \in \mathit{reachable}(e) \quad x \in \mathit{reachable}(K_1)$$
$$\langle\!\langle H \mid y \mid K_2\rangle\!\rangle\ \mathbf{using}\ x \leftarrow \langle\!\langle H \mid e \mid K_0\rangle\!\rangle,\ y \leftarrow \langle\!\langle H \mid x \mid K_1\rangle\!\rangle$$

This solution works only if all update frames that are residualized have a representable type. We can not guarantee representability of these frames without imposing inconvenient constraints on the input. An alternative solution (at the cost of losing some sharing) is to rebuild (onto the heap) the expressions corresponding to stackframes downto the critical update frame. The *rebuild* function takes a stack segment and a variable, and rebuilds a term for the heap by undoing evaluator steps that produce stack frames. In combination with a more concrete application splitting example it looks like the following:

$$\langle H \mid f \mid \mathbf{apply}\ a, \mathbf{apply}\ b, K_0, \mathbf{update}^t\ r, K_1\rangle \quad r \in (\mathit{reachable}(a) \cup \mathit{reachable}(b))$$
$$\langle\!\langle H \mid z \mid K_0, \mathbf{update}\ r, K_1\rangle\!\rangle\ \mathbf{using}\ x \leftarrow \langle\!\langle H, r \overset{t}{\mapsto} \mathit{rebuild}\ K_0\ z \mid a \mid \epsilon\rangle\!\rangle,\ y \leftarrow \langle\!\langle H, r \overset{t}{\mapsto} \mathit{rebuild}\ K_0\ z \mid b \mid \epsilon\rangle\!\rangle,\ z \leftarrow f\ x\ y$$

Splitting case expressions When the top of the stack is a case scrutiniser frame, the splitter produces a residual case expression with a substate for each case alternative. Each alternative gets the whole rest of the stack in its substate, and the matched constructor or literal is added as a heap binding with a default tag. Copying the the rest of stack is required to not block further optimizations, however this can cause significant code size growth during supercompilation.

$$\langle H \mid x \mid \mathbf{scrutinise}\ \{C\ a\ b \to p; D\ c \to q\}, K\rangle$$
$$\mathbf{select}\ x\ \mathbf{with}\ C\ a\ b \rightarrowtail \langle\!\langle x \overset{t0}{\mapsto} C\ a\ b, H \mid p \mid K\rangle\!\rangle,\ D\ c \rightarrowtail \langle\!\langle x \overset{t0}{\mapsto} D\ c,\ \ H \mid q \mid K\rangle\!\rangle$$

3.6 Adding types

The operational semantics and thus the evaluator need to be extended with the types from System FC[15]. Because hardware has only monomorphic types, the matcher and generalization do not need special treatment for types. The rest is straightforward bookkeeping. The NETCORE language is also typed, because size information of in/outputs is required to convert to other netlist formats.

Splitting case with GADTs The splitter needs to check the types for each alternative to determine if that alternative is reachable. Unreachable alternatives are removed before splitting the case expression. Removal of unreachable alternatives is crucial for the termination of the splitter, for example with recursively (as GADT) defined fixed sized vectors.

```
data Vec :: * -> * -> * where
  VNil  ::                    Vec Z a
  VCons :: a -> Vec n a -> Vec (S n) a

vectFold :: (a -> b -> b) -> b -> Vec n a -> b
vectFold f z VNil         = z
vectFold f z (VCons x xs) = f x (vectFold f z xs)

sumV5 :: Vec Nat5 Int -> Int
sumV5 = vectFold (+) 0
```

3.7 Properties of this supercompiler

Proving a supercompiler correct would be a whole paper on itself, so we will give only some idea why it would be correct. Firstly we assume that the description in Haskell makes sense as a hardware component and all the restriction that involves. Secondly we assume that supercompilation is only attempted after the descriptions have been run (without errors) as Haskell program for all possible inputs.

Completeness of the translation can be shown by the totality of the splitter function combined with the termination of the whole process. The splitter is not defined for all syntacticly possible inputs, so it must be shown that these inputs will not happen. Well-typed programs don't get stuck, thus evaluation should only block on external inputs or components. The splitter must delay component instantiation to the last possible moment, and must pass on all possible information to the substates.

Termination of the supercompiler is derived from the fact that the splitter produces only strictly smaller sub states. And rest of termination argument is based on the assumption that the input is sensible hardware (finite, fixed size input, etc.).

That the supercompilation process preserves the semantics of the described hardware is fairly strait forward. The evaluator and matcher are directly derived from operational semantics. And for the splitter it can be proven by using local case analysis.

3.8 Disadvantages of supercompilation

Structure present in the input is lost in the generated hardware. This can make lowlevel debugging harder and might affect optimizations heuristics of synthesis tools.

Supercompilation is sometimes too aggressive causing exponential blowup in code size. In these naive bitCount and bitReverse implementations the supercompiler produces a very big tree like lookup structure that yield the result for each possible input directly.

```
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

bitReverse :: Word32 -> Word32
bitReverse x = br x 0 nat32 where
  br _ b Z     = b
  br a b (S c) = br (a `shiftR` 1)
    ((if even a then 0 else 1) + 2*b) c
```

In the bitCount example the splitter extends the branch over the addition and the recursive call, thus producing a big tree. And in the bitReverse example the problem is that the br function is unintendedly specialized over its second argument. This problem is probably not solvable in the supercompiler itself, because distinguishing the compile time part from the intended hardware part is undecidable. And retroactively undoing the spurious specializations its not always feasible, because the supercompiler might run out of memory before that.

These problems only seems to occur in combination of a finite recursion that branches on an input. In all example we have found so far the problem is easy to avoid by writing slightly different code, for these examples it is using a getBit function instead.

4. Extensions for sequential hardware

Until now we only considered combinatorial hardware descriptions. For less trivial hardware components, we need to describe things with state (delays, registers and memory blocks).

Streams as signals We choose streams to implement signals, because they offer an elegant, simple and executable notation for the implementation of synchronous systems. A large part of their elegance is that streams are applicative functors.

```
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)
```

4.1 Supercompiling streams

After supercompilation we want to have only stream functions of a specific form: Single-recursive functions that pattern matches on all input streams and results in a stream constructor. And the recursive call is on all tails of all input streams. Example:

```
foo (a:<as) (b:<bs) (c:<cs) =
   x <- bar a b c
   xs <- foo as bs cs
 x :< xs
```

We rely on fusion capabilities of supercompilation to produce only stream functions of the directly recursive form.

Evaluating streams The evaluation state is extended by a stream context S, which is a sequence of stream bindings ($a :< as \leftarrow bs$).

$$\langle H \mid S \mid e \mid K \rangle$$

The stream context is extended when trying to evaluate a case of the stream constructor on a free variable.

$$\textsc{Stream}\ \langle H \mid S \mid \mathbf{case}\ x\ \{a :< as \to e\} \mid K \rangle \rightsquigarrow$$

$$\langle H \mid a :< as \leftarrow x, S \mid e \mid K \rangle \ \textit{(x is free)}$$

All other evaluation rules keep the stream context unmodified. The reason to extend the evaluation state is to be able to continue the evaluation until we find a stream constructor value. So that all the matching on the input streams and the recursive stream function call has been exposed at that point.

Splitting of streams Splitting of a stream constructor is similar to other constructors, except that the whole stream context is removed and residualized. Thanks to previous evaluator extension the splitter can produce the output in the expected format directly.

$$\langle H \mid \overline{a :< as \leftarrow bs} \mid x :< xs \mid \epsilon \rangle$$

$$\overline{a :< as \leftarrow bs}\ \mathbf{in}$$
$$y :< ys\ \mathbf{using}$$
$$y \leftarrowtail \langle\!\langle H \mid \epsilon \mid x \mid \epsilon \rangle\!\rangle$$
$$ys \leftarrowtail \langle\!\langle H \mid \epsilon \mid xs \mid \epsilon \rangle\!\rangle$$

In all other cases splitting just passes on the stream context to its substates.

4.2 Stateful components

A small library of primitive functions for delay, registers and memory block would suffice for most hardware designs. We choose not to use builtin primitives for state, because it poses interesting challenges for the supercompilation algorithm. And it allows for describing state machines as (multiple) recursive functions over streams.

```
delay :: a -> Signal a -> Signal a
delay x ~(y :< ys) = x :< delay y ys

mealy :: (s -> i -> (s,o)) -> s ->
  Signal i -> Signal o
mealy f s ~(i :< is) = o :< mealy f s' is
  where (s', o) = f s i
```

For example a delay combinator can be written as a function over streams with as extra argument the thing to be delayed. And this mealy machine has a transition function as first argument and the internal state as the second. We need to use lazy pattern matching on streams here because such stateful components can be a part of a feedback loop. The extra state arguments on stream functions may only be used with a type that is representable in hardware.

Lazy pattern matching on streams GHC translates lazy pattern matches to let bindings with case expression, so they end up as heap bindings. To produce the same normal form with lazy stream matches, we need to make them strict during evaluation. When evaluation blocks on a stream constructor value, these case expressions are evaluated to expose stream matches within them.

4.3 Supercompiling stateful streams

Accumulating state arguments causes the supercompilation process to not terminate anymore. For example:

```
acummulator :: Signal Int -> Signal Int
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: $\langle a_0 \overset{t3}{\mapsto} 0, y_0 \overset{t5}{\mapsto} a_0 + b_0, ys_0 \overset{t6}{\mapsto} accum\ y_0\ bs_0$
$\quad \mid b_0 :< bs_0 \leftarrow xs_0 \mid y_0 :< ys_0 \mid \epsilon\rangle$
S^2: $\langle a_0 \overset{t3}{\mapsto} 0, y_0 \overset{t5}{\mapsto} a_0 + b_0, y_1 \overset{t5}{\mapsto} y_0 + b_1, ys_1 \overset{t6}{\mapsto} accum\ y_1\ bs_1$
$\quad \mid b_1 :< bs_1 \leftarrow bs_0 \mid y_1 :< ys_1 \mid \epsilon\rangle$
S^3: $\langle a_0 \overset{t3}{\mapsto} 0, y_0 \overset{t5}{\mapsto} a_0 + b_0, y_1 \overset{t5}{\mapsto} y_0 + b_1, y_2 \overset{t5}{\mapsto} y_1 + b2, \ \ldots$
$\ldots$

Because of the growing evaluation states, the matcher can not find a state to tie back to, so it will not terminate. The solution to this problem is to avoid the infinite state argument accumulation. We use the growing tag heuristic from Bolingbroke to detect the problematic heap binders.

For every evaluation state, a bag with all tags on the heap binders is collected. When the supercompiler reaches S^2 it finds a previous state S^1 with the same set of tags. The tag $t5$ is has a greater multiplicity in S^2 than in S^1, so the heap binders corresponding to that tag are to blame for potential nontermination. The supercompilation process is rolled back to state S^1 and the binders with tag $t5$ are split off and supercompiler resumes with the rest of the state. Checks for rollback are only done if the focus term of the evaluation state is a stream constructor. Disabling sharing of representable heap binders when in a stream function context helps accumulator detection.

Supercompiling the same example with this rollback technique produces the following output (after inlining):

```
acummulator :: Signal Int -> Signal Int
acummulator (b0 :< bs0) = let y = 0 + b0 in
  y0 :< f2 y0 bs0

f2 y0 (b1 :< bs1) = let y1 = y0 + b1 in
  y1 :< f2 y1 bs1
```

The problem of the recursion being unpeeled is addressed later.

4.4 Improving rollback

Figure 6 shows the toplevel code of the supercompiler with both forms of rollback. This code does not include the bookkeeping for the unique name supply, types and cleaning up of the supercompilation state when rolling back.

Making rollback more precise The elegant tagbag method for termination checking and rollback has been shown to be both efficient and effective. [4, 6] However, the effectiveness of the tagbag method relies on the fact that each tag is relatively rare. This breaks down when a lot of abstractions are used in a program which give rise to a lot of similar polymorphic expressions. In case of hardware descriptions, the use of applicative functors on streams is very common, which contains a polymorphic function application in its implementation.

We can try to make the tags more precise by adding extra information, for example using the first type constructor in type of the tagged expression. This workaround helps a bit, but is not good enough for larger hardware descriptions as they tend to use a small set of types in the definition of a lot of other types. We tried other methods to add more information to the tags, but failed to

```
type Hist = [(String, EvalState, Maybe TagBag)]

data RollBackException
 = RollBackAccumExp String (Set Name)
 | RollBackInitExp String (Set Name)
 | RollBackInitVar String EvalState Renaming

checkForInits :: String -> Hist -> SCM ()
checkTagsAccum :: TagBag -> EvalState ->
    Hist -> SCM ()
splitOffBinders :: Set Name -> EvalState ->
  Bracket

sc :: Hist -> TypedFVs -> EvalState ->
  SCM Instantiation
sc h fvts s = do
 let state = simplify (reduce s)
 let inSS = hasStreamResultType state
 let hist = if inSS then h else []
 let fvs = stateFVs state
 ps <- gets promises
 case findMatchingState state ps of
  Just (q, rn) -> do
   checkForInits (funName q) hist
   return (CompInst (funName q)
     (map (rename rn) (freeVars q)))
  Nothing -> do
   let tb = heapTagBag state
   when (isStreamConExp state)
     (checkTagsAccum tb state hist)
   n <- promise fvs state
   let bracket = split state
   let mtb = if isStreamConExp state
     then Just tb else Nothing
   let hist' = if inSS
     then (n, state, mtb) : hist else hist
   (do
      ys <- mapM (sc hist') (holes bracket)
      bind n fvs (assemble bracket ys)
   ) `catchError` (\rbe -> case rbe of
       RollBackAccumExp nx aes
         | n == nx -> do
          let rbr = splitOffBinders aes state
          let rh = (n, state, Nothing) : hist
          ys <- mapM (sc rh) (holes rbr)
          bind n fvs (assemble rbr ys)
         _ -> throwError rbe
   ) `catchError` (\rbe -> case rbe of
       RollBackInitExp nx ies
         | n == nx -> do
          let rbr = splitOffBinders ies state
          ys <- mapM (sc hist) (holes rbr)
          bind n fvs (assemble rbr ys)
       RollBackInitVar nx gstate rn
         | n == nx -> do
          let ys = varMappingsFromRN rn
          z <- sc hist gstate
          bind n fvs (CompBody ys z)
       _ -> throwError rbe
   )
   return (CompInst n fvs)
```

Figure 6. Code for supercompiler toplevel with rollbacks

solve the problem completely and found calculating the tagbag for such solutions very expensive.

Heuristic for selecting what to rollback The solution lies in another property that accumulators have in hardware descriptions. They are exposed as heap binders that exist in both current and previous evaluation states, so we can use that extra information to split them off. Because checking for accumulators is only done on stream constructor expressions, the same binder in both means that it has been carried over from one cycle to the next. We can use the names of heap binders implicitly as heap locations, because a new name is generated only when allocating a let-expression onto the heap. Combining the growing tag heuristic with cross cycle binder detection gives a cheap and very effective rollback technique. The only potential problem is that certain constant expressions could be split off if they happen to have the same tag as an accumulator. For general purpose supercompilation, filtering on old binders in the growing tag heuristic could be a nice improvement.

Avoiding initial state propagation Another reason that stream functions can be unpeeled is specialization on initial states. For example, a simple component that averages the last four inputs:

```
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, because *avg4* is specialized on the zeros.

```
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
```

This initial state propagation can be avoided by adding another form of rollback. When the matcher finds a recursion in the stream function, the supercompilation history is searched for the earliest evaluation state that is an instance of the matched state. Then the heap bindings that specialize this instance of the evaluation state are split off and the supercompilation process is restarted from here. Initial-state rollback can override an acummulation rollback.

Initial-state specialization by variables only This can also happen with an initial state that only differs in using less variables than a later state. We handle this by rolling back to the initial state and then resuming supercompilation with the renamed late state. A similar problem has been described in section 6.2.5 of Max Bolingbroke's PhD thesis.

5. From supercompiled core to hardware

Supercompilation has removed all abstractions, but a few post processing steps are required before hardware can be generated. The first step is that of inlining all those functions that are trivial or used only once. Supercompilation produces such functions in great numbers.

Lambda floating A lambda represents a component with an input, but supercompilation can leave lambdas inside case alternatives. We thus need to float lambdas out of case statements (in a bottom up way), until only the outermost expressions are lambdas.

$$\begin{array}{l}\textbf{select } x \textbf{ with} \\ \quad C\ a\ b \rightarrowtail \lambda p.e \\ \quad D\ c \rightarrowtail \lambda q.f\end{array} \quad \Longrightarrow \quad \begin{array}{l}\lambda p.\ \textbf{select } x \textbf{ with} \\ \quad C\ a\ b \rightarrowtail e \\ \quad D\ c \rightarrowtail f[q/p]\end{array}$$

5.1 Post processing stream functions

Stream binders behave like lambdas and can end up inside case alternatives, so we need to float them out like we did with the lambdas.

Dealing with state machines State machines are ubiquitous in hardware descriptions. Quite often, multiple recursive stream functions representing state machines can be merged into a single one by introducing a datatype with one constructor per function, where the arguments of each constructor are the state arguments of the corresponding function. Consider as an example:

```
pulseLengthCounter xs = countPulse 0 xs
noPulse (x:<xs) = case x of
  Low  -> 0 :< noPulse xs
  High -> 0 :< (countPulse 1) xs
countPulse n (x:<xs) = case x of
  Low  -> n :< noPulse xs
  High -> n :< (countPulse (n+1)) xs
```

which can be converted to

```
data StateX = NoPulse | CountPulse Word16
pulseLengthCounter xs = foo (CountPulse 0) xs
foo s (x:<xs) = case s of
   NoPulse -> case x of
     Low  -> 0 :< foo  NoPulse xs
     High -> 0 :< foo (CountPulse 1) xs
   CountPulse n -> case x of
     Low  -> n :< foo  NoPulse xs
     High -> n :< foo (CountPulse (n+1)) xs
```

If the heuristics for rollback fail to produce a single recursive function, we can resolve that by using this state machine conversion.

Implicit state machines It is possible to hide state in a single recursive stream function by reordering the stream arguments:

```
swapSub :: Signal Bool -> Signal Int ->
  Signal Int -> Signal Int
swapSub (c:<cs) (x:<xs) (y:<ys) =
  (x - y) :< if c
    then swapSub cs ys xs
    else swapSub cs xs ys
```

The state in such cases can be made explicit by creating specialized copies of each stream function that is applied with a varying order of stream arguments. After that we can apply the same method of dealing with state machines described previously.

Removing all streams and introducing delays When all stream functions are in direct recursive form, we can remove all stream constructs and insert the delay primitive for state.

$$\begin{array}{l} foo\ as\ bs \leftrightharpoons \\ \quad foo2\ as\ bs\ x\ y\ \textbf{using} \\ \quad\quad x \leftharpoonup constX \\ \quad\quad y \leftharpoonup constY \\ foo2\ (a :< as)\ (b :< bs)\ x\ y \leftrightharpoons \\ \quad c :< c\ \textbf{using} \\ \quad\quad c \leftharpoonup bar\ a\ b\ x\ y \\ \quad\quad x' \leftharpoonup fx\ a\ b\ x \\ \quad\quad y' \leftharpoonup fy\ a\ b\ y \\ \quad\quad cs \leftharpoonup foo2\ as\ bs\ x'\ y' \end{array}$$

is converted to the following streamless form:

$$
\begin{array}{l}
foo\ a\ b \leftrightharpoons \\
\quad bar\ a\ b\ x\ y\ \textbf{using} \\
\qquad x \leftarrow delay\ constX\ x' \\
\qquad x' \leftarrow fx\ a\ b\ x \\
\qquad y \leftarrow delay\ constY\ y' \\
\qquad y' \leftarrow fy\ a\ b\ y
\end{array}
$$

5.2 Undoing duplication from supercompilation process

Duplication is caused by cloning the heap for subcomponents during splitting, and/or by copying the stack when splitting case expressions.

This is not a critical problem, because it mostly causes the same component to be used multiple times and does not affect code size. The matcher finds identical components during supercompilation. For the stack, the duplication is limited by logic depth. When instantiating supercompiled hardware exponential blowup in size is possible.

Recovering sharing Identical components are easy to detect due to matching during supercompilation. We need to use a bottom up transformation to merge identical components, because instantiating and flattening all components first could blow up exponentially. The algorithm for undoing the duplication caused by loss of sharing is:

1. Sort all components by the maximum nesting depth of components inside them
2. For all components, starting with the deepest one:
   (a) Float 'free' components out of case expressions
   (b) Instantiate components that cannot be floated out
   (c) Unify the component instantiations with identical inputs
   (d) Update the current component with the result

Pushing case over common expressions Due to stack duplication when splitting case expressions, residual case alternatives can be similar, differing only in some subexpression. In the following example, the multiplication with x can be floated out of the case alternatives:

```
case c of
  True  -> 3 * x
  False -> 5 * x

(case c of {True -> 3; False -> 5}) * x
```

Duplication by splitting of recursive lets The rebuilding of the stack onto the heap in the splitter rule for recursive lets may cause duplication of components with multiple feedback loops. Figure 7 shows a graphical representation of an example with both the produced and the expected output. This could be solved by some extensive post processing the resulting netlist. However, this code size explosion (exponential in the number of feedback loops) could make the supercompiler run out of memory before post processing. The workaround is amending the splitter to not duplicate in specific cases.

$$\langle H \mid e \mid K_0, \mathbf{update}^t\ x, K_1 \rangle$$
$$x \in fvs(e)$$

(x is a signal with a product type of representable elements)

$$
\begin{array}{l}
\langle\langle H \mid x \mid K_1 \rangle\rangle\ \textbf{using} \\
\quad x \leftarrow \langle\langle H \mid e \mid K_0 \rangle\rangle
\end{array}
$$

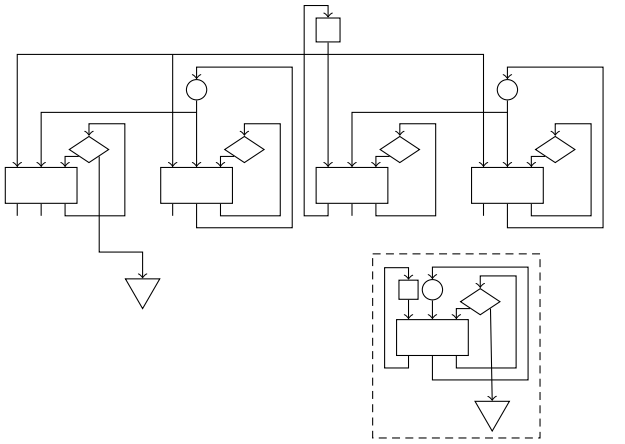

Figure 7. Example of duplication by splitting of recursive lets

We apply this special splitter rule only on feedback loops on components with multiple outputs. This heuristic might have an effect on completeness of the translation process.

5.3 From datatypes to bitvectors

Custom encodings are important for interfacing with existing components and for very compact encodings. This allows hardware designs for existing instruction sets or protocols to be specified using nice datatypes, instead of bitvectors.

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

Given this class, we can define a standard translation to remove all data constructors. First, the optimized lowlevel hardware description is converted back into the simple core language. Then the *encode* and *decode* functions are inserted around each constructor and case expression.

```
Foo a b ⇛ encode (Foo (decode a) (decode b))

case x of
  Foo a b -> e
  Bar c   -> f

      ⇛

case decode x of
  Foo a' b' ->
    let a = encode a'
    let b = encode b'
    in e
  Bar c'  -> let c = encode c' in f
```

After that we go through the whole supercompilation process again, until we arrive again at the same lowlevel representation, but this time without any constructor or case expression.

The reason to not insert en/decode functions before the first supercompilation pass is efficiency. If all case expressions are converted to their bitlevel variants, then supercompilation could have to deal with over ten times as much code. Another reason is that not all datatypes have a bitvector encoding; some only exist as generative abstractions at compile time.

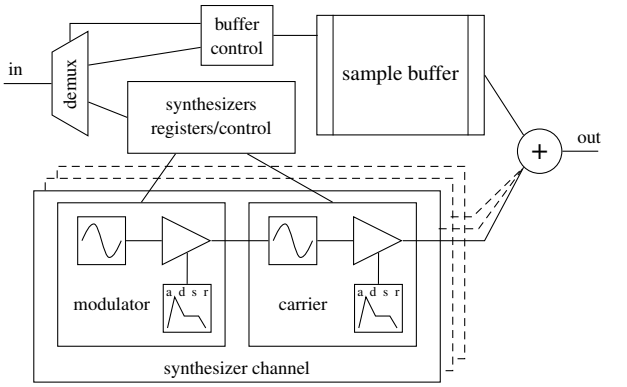

Figure 8. Overview of the soundcard components

6. Soundcard chip example

Here is an example to show how supercompilation enables more expressiveness in describing hardware. We chose to describe a simplified digital soundcard with early 90s features: a synthesizer with multiple channels using frequency/phase modulation, a buffer for direct sound sample output and a simple instruction set to control the whole thing. Figure 8 shows the structure of this design.

The first component is oscillator that produces various sine waves using compile time generated lookup tables. To make the lookup table smaller the waveforms are synthesized using symmetry from a quarter sine table.

```
data WaveForm = FullSine | AbsoluteSine
  | HalfSine | QuarterSine deriving (Eq, Enum)

oscillator:: WaveForm -> Word16 -> Word16 -> Int16
  -> (Word16, Int16)
oscillator wform fi x mods = (x', amplitude) where
  x'      = x + fi
  modx    = x + toW16 mods
  offset  = (modx `shiftR` 4) .&. 0xFF
  quad    = (modx `shiftR` 14) .&. 0x3
  index   = if odd quad then 255-offset else offset
  qSin    = lookupTable 256 qSinTable index
  amplitude = waveSelect quad wform qSin

qSinTable = [round (sin ((x+0.5)*2*pi/1024)*32767)
  | x <- [0..255]]

waveSelect 0 _           x = x
waveSelect 1 QuarterSine _ = 0
waveSelect 1 _           x = x
waveSelect _ FullSine    x = negate x
waveSelect _ AbsoluteSine x = x
waveSelect _ _           _ = 0
```

The second component is an envelope generator, allowing the amplitude of a played tone to change over time.

```
data EnvelopeMode = Attack | Decay
  | Sustain | Release deriving Eq

envelope mode x a d s r = case mode of
  Attack  | x+a < x  -> (Decay  , maxBound)
          | otherwise -> (Attack , x + a)
  Decay   | x-d < s  -> (Sustain, s      )
          | otherwise -> (Decay  , x - d)
  Sustain             -> (Sustain, s)
  Release | x-r > x  -> (Release, 0      )
          | otherwise -> (Release, x - r)
```

Next we connect the oscillator and envelope generator with an amplifier to form an operator. An extra input allows the operator to be modulated by another signal. The operator includes the register and delay elements, and is written an applicative style. We postpone the decision for space/time tradeoff with multiple operators by parametrizing the stateful components.

```
data OperReg = WaveForm | Volume | FreqStep
  | EnvReg EnvelopeMode deriving Eq

operator :: (forall a.a -> Signal a -> Signal a)->
  (forall b.Enum b => OperReg -> b -> Signal b) ->
  Signal Bool -> Signal Int16 -> Signal Int16
operator delay ctrlReg play ms = out where
  pps       = delay False play
  wf        = ctrlReg WaveForm FullSine
  fi        = ctrlReg FreqStep 10
  (x',osc)  = unpack (liftA4 oscillator wf fi x ms)
  x         = resetOsc<$>play<*>pps<*>delay 0 x'
  resetOsc p pp y = if p && not pp then 0 else y
  [a,d,s,r] = [ctrlReg (EnvReg e) i | (e,i) <-
    [(Attack,5),(Decay,2),(Sustain,0),(Release,1)]]
  env'      = liftA6 envelope em gain a d s r
  (em',gain) = unpack $ delay (Release,0) env'
  em        = toggleEnv<$>play<*>pps<*>em'
  out = amplify<$>osc<*>gain<*>ctrlReg Volume 64

toggleEnv True  False _ = Attack
toggleEnv False True  _ = Release
toggleEnv _     _     e = e

amplify x gain vol = toI16 (res `shiftR` 16) where
  res = toI32 x * toI32 (shiftR gain 8) * toI32 vol
```

The toplevel of this soundcard connects the output of the 8 synthesizer channels and the sample buffer with a mixer. State machines are used to: split the input into commands and raw samples and to control the read side of the sample buffer.

```
soundcard :: Signal Int16 -> Signal (Maybe Int16)
soundcard ins = mixer 0 chan xs ys where
  chan = moduloCounter 8
  (cmd,wrS) = unpack (demuxCommand ins)
  xs = synthesizers 8 chan cmd
  ys = sampleBuffer (bufferCtrl cmd chan) wrS
  sampleBuffer = regFile 65536 0

mixer accum (0:<chs) (x:<xs) (y:<ys) =
  Just (shiftR accum 3) :< mixer (y+x) chs xs ys
mixer accum (_:<chs) (x:<xs) (y:<ys) =
  Nothing :< mixer (accum+x) chs xs ys

bufferCtrl = setCtrl 0 0 where
  setCtrl nr ri (r:<rs) cs = case r of
    PlayIndex i  -> i  :< rdUpd nr i rs cs
    PlaySamples s -> ri :< rdUpd s ri rs cs
  rdUpd nr ri rs (c:<cs)
    | nr/=0 && c==0 = setCtrl (nr-1) (ri+1) rs cs
    | otherwise     = setCtrl nr     ri     rs cs

demuxCommand = recCmd 0 where
  recCmd wi (x:<xs) = case decode (toBV x) of
    WriteIndex i -> (Nop,Nothing):<recCmd i xs
    WriteBuff n  -> (Nop,Nothing):<recSample n wi xs
    c            -> (c  ,Nothing):<recCmd wi xs
```

```
recSample n i (x:<xs) = case n of
  0 -> (Nop, Just (i,x)):<recCmd (i+1) xs
  _ -> (Nop, Just (i,x)):<recSample (n-1) (i+1) xs

type Channel = Word8
type Octave  = Word8
data Tone = Tone Octave Note
data Note = A | Ash | B | C | Csh | D | Dsh
  | E | F | Fsh | G | Gsh deriving Enum

data OperType = Carrier | Modulator deriving Eq
data Command = Nop | SetMRatio Channel Word8
  | WriteIndex Word16 | WriteBuff Word16
  | PlayIndex Word16  | PlaySamples Word16
  | SetReg Channel OperType OperReg Word8
  | PlayTone Channel Tone | StopTone Channel

instance Representable Command where
  type BitSize Command = D16
  decode cv = ... -- omitted for space reasons
```

Finally we construct the set of synthesizer channels from the operators. Because of the low frequency of audio signals all channels share the same synthesizer by time multiplexing. Connecting all register files and converting the commands to register write signals takes a fair bit of code.

```
synthesizers :: Channel -> Signal Channel ->
  Signal Command -> Signal Int16
synthesizers nCh chan cmd = out where
  (rMR, wrMR, wrP) = unpack (chRegs <$> cmd)
  modR = regFile nCh 0 rMR wrMR
  (wrM, wrC) = unpack (operCtrl<$>cmd<*>modR)
  play = regFile nC False chan wrP
  xDelay ini = delayFile nCh ini chan
  ctrlReg wr key ini = regFile nCh ini chan
    (fmap (fmap fromW16) . wrSel key <$> wr)
  mS = operator xDelay (ctrlReg wrM) play (pure 0)
  out = operator xDelay (ctrlReg wrC) play mS

wrSel::Eq k => k-> Maybe ((k,i),x) -> Maybe (i,x)
wrSel kA (Just ((kB,i),y)) | kA==kB = Just (i,y)
wrSel kA _ = Nothing

chRegs (SetMRatio c r) = (0, Just (c,r), Nothing)
chRegs (PlayTone c k) = (c,Nothing, Just (c,True))
chRegs (StopTone c)  = (0,Nothing, Just (c,False))
chRegs _ = (0, Nothing ,Nothing)

topAStep = 440*2^3 * 2^16/44100
noteSteps = map (\x -> round (topAStep*2**(x/12)))
  [0..11]

operCtrl (SetReg ch op reg x) _ = demux wrx where
  demux w = (wrSel Modulator w, wrSel Carrier w)
  wrx = Just ((op,(reg,ch)),fixRVal reg (toW16 x))
  fixRVal (EnvReg Sustain) x = x `shiftL` 8
  fixRVal (EnvReg _      ) x = shiftR (x^2) 2 + 1
  fixRVal _                x = x
operCtrl (PlayTone c (Tone oct note)) modR =
 (wrFStep mf, wrFStep cf) where
  wrFStep x = Just ((FreqStep, c), toW16 x)
  cf = shiftR nstep (7-fromEnum oct)
  mf = shiftR (nstep*toW32 modR) (12-fromEnum oct)
  nstep = lookupTable 12 noteSteps (fromEnum note)
operCtrl _  _ = (Nothing, Nothing)
```

7. Comparison

Common hardware description languages, such as VHDL and Verilog have very limited power of generative abstractions. Even worse for expressiveness, the synthesizable subset often depends on the tooling being used.

7.1 Hardware Description EDSLs

There are multiple domain specific languages for hardware design embedded in Haskell, such as the well-known Lava [3] family of libraries. A so-called deep embedding is used to synthesize a DSL description to a netlist. That is, the DSL primitives are constructors for a netlist graph datatype; synthesis of a description composed of these primitives is simply the calculation/construction of the Haskell expression. Techniques such as *observable sharing* [8] must be used to detect cycles and so ensure termination of the netlist generation.

The rich set of *choice*-constructs in Haskell, such as pattern matching, cannot be reflected down to the eventual netlist when using the EDSL approach. This is a consequence of using Haskell's evaluation mechanism to construct the netlist graph: choice constructs can be used to guide the construction, but it is not possible / feasible to *observe* all the alternatives. A developer using an EDSL can hence only use choice-constructs offered by the EDSL library to represent choice in the netlist. Supercompilation can however observe Haskell's choice-constructs in their entirety, meaning that a developer can use these elements to describe choice in the netlist.

In the long term, we expect Haskell to support a way to deeply embed most language constructs (where pattern matching and guards are the crucial aspects). This will benefit all kinds of EDSLs, and will minimize the difference (from the user's point of view) between the EDSL approach and supercompilation for hardware descriptions.

7.2 Current Transformations in CλaSH

The compiler for CλaSH uses a set of simple rewrite steps that are based on the syntax and types of the input expressions. Supercompilation as described in this work is however based on the semantics of the expressions.

The CλaSH compiler uses a combination of inlining and specialisation to transform a higher-order function hierarchy to a first-order one, but leaves the function hierarchy intact otherwise. It is therefore easier to reason about the resulting netlist, as there is an almost one-to-one correspondence between the component-hierarchy and the function-hierarchy.

A disadvantage of the current CλaSH compiler is that the set of transformations cannot unroll static recursion, meaning that recursive expressions cannot be transformed into a netlist. Additionally, it is hard to prove that a rewriting system such as the one used by the CλaSH compiler has a canonical first-order normal form. Concisely defining to which restrictions an input program must adhere for it to be synthesizable to a netlist is therefore non-trivial.

7.3 Partial Evaluation

Supercompilation and partial evaluation, both being symbolic evaluation mechanisms, have a lot in common; so much so that it hard to quantify the difference. Existing partial evaluation techniques are however not powerful enough to perform the stateful stream fusion as performed by the supercompilation process described in this work.

The compiler for the Bluespec [12] hardware description language makes extensive use of partial evaluation to completely unroll recursive function definitions, and reduce the function hierarchy to a first-order form. There is however no account of the exact details of its partial evaluation mechanism, nor an exhaustive list of restrictions / requirements on the input programs.

8. Conclusions

We have proposed an alternative way to produce hardware from a high level language, which trades predictability of the resulting hardware for increased expressiveness. The result is that the full input language is available for hardware design and could be applied to CλaSH and lava style hardware descriptions. The use of streams offers a good basis for (local) state abstractions.

Hardware design, functional programming and supercompilation are a perfect triple. Extending the perfect match from Sheeran [14]:

- Direct correspondence combinatorial hardware and functional languages
- Hardware domain avoids supercompilation disadvantages
- Functional languages are good at generative abstractions
- Supercompilation is easier with simple semantics
- Translation to netlist $\simeq$ removing abstraction $\simeq$ unconstrained Supercompilation

Supercompilation is the sledgehammer approach to hardware description: guaranteed success but the results might not be recognizable. Although it is powerful, it is inflexible in the sense that the semantics of the source language determines the produced hardware.

Any high level language to hardware Unconstrained supercompilation could, in principle, be used to translate any language to the netlist level, as nothing in the presented technique is Haskell specific; it merely requires an input language with operational semantics and an output language that can be synthesized to hardware. Simplicity of the operational semantics benefits the whole supercompiler. The output language is preferably close to a subset of the input language. How to model sequential and stateful components in a executable manner is an important choice to make.

The evaluator for the input language is just an implementation of the operational semantics. A matcher for the eval-state, follows from state structure and variable naming implementation. A simplifier that includes garbage collection, and maybe some more transformations to make the splitter simpler. The splitter is the critical part, however most choices are determined by structure of the eval-state and the output language. It might be necessary to modify the output language or the evaluator to make the implementation of the splitter feasible. Some further post processing and optimizations could be applied to make the generated hardware more efficient or to overcome differences between the in- and output languages.

8.1 Future work

This work is now at the proof of concept stage and needs more work before a wider audience can use it as a practical tool. Besides documentation we need a comprehensive library for hardware components. A decision has to made on which type level numeral library to support. Iavor Diatchki's work on integrating type level natural in GHC looks promising. Also warning the user on accidently infinite hardware descriptions would be useful, and can be done by integrating a very relaxed variant of termination criteria proposed in other works.

Creating formal proofs of the properties of this supercompiler requires further research. And making the supercompilation process faster is long term concern, especially with the iterative rollback procedure we introduced.

Dependently typed languages for hardware The total nature of hardware matches well with dependently typed languages. And the ubiquitous use of fixed size data types requires a lot of manipulation of type level numerals. Plus the verification of hardware could be assisted by encoding more properties into types. Thus adapting this work to a language like Agda or Idris would be interesting.

Acknowledgments

This research is conducted within the Modern project(647.000.003) supported by NWO. We thank Max Bolingbroke for making available the source code for his supercompiler and discussions on supercompilation in general.

References

[1] E. Axelsson, K. Claessen, and M. Sheeran. Wired: Wire-Aware Circuit Design. In *Proceedings of Conference on Correct Hardware Design and Verication Methods (CHARME)*, volume 3725 of *Lecture Notes in Computer Science*, pages 5–19. Springer Verlag, 2005.

[2] C. P. R. Baaij, M. Kooijman, J. Kuper, W. A. Boeijink, and M. E. T. Gerards. CλaSH: Structural Descriptions of Synchronous Hardware using Haskell. In *Proceedings of the 13th Conference on Digital System Design*, pages 714–721, USA, September 2010. IEEE Computer Society.

[3] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware Design in Haskell. In *Proceedings of the third International Conference on Functional Programming (ICFP)*, pages 174–184. ACM, 1998. ISBN 1-58113-024-4. doi: 10.1145/289423.289440. URL `http://doi.acm.org/10.1145/289423.289440`.

[4] M. C. Bolingbroke. *Call-by-need supercompilation*. PhD thesis, University of Cambridge, May 2013.

[5] M. C. Bolingbroke and S. L. P. Jones. Supercompilation by evaluation. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Haskell*, pages 135–146, 2010.

[6] M. C. Bolingbroke and S. L. P. Jones. Improving supercompilation: tag-bags, rollback, speculation, normalisation, and generalisation. unpublished, March 2011. URL `www.cl.cam.ac.uk/~mb566/papers/chsc2-icfp11.pdf`.

[7] M. C. Bolingbroke, S. L. P. Jones, and D. Vytiniotis. Termination combinators forever. In K. Claessen, editor, *Haskell*, pages 23–34. ACM, 2011. ISBN 978-1-4503-0860-1.

[8] A. Gill. Type-Safe Observable Sharing in Haskell. In *Proceedings of the second Haskell Symposium*, pages 117–128. ACM, Sep 2009. ISBN 978-1-60558-508-6. doi: http://dx.doi.org/10.1145/1596638.1596653.

[9] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation*. Prentice Hall international series in computer science. Prentice Hall, 1993. ISBN 978-0-13-020249-9.

[10] J. Matthews, B. Cook, and J. Launchbury. Microprocessor specification in Hawk. In *Proceedings of 1998 International Conference on Computer Languages*, pages 90–101, May 1998. doi: 10.1109/ICCL.1998.674160.

[11] N. Mitchell. Rethinking supercompilation. In P. Hudak and S. Weirich, editors, *ICFP*, pages 309–320. ACM, 2010. ISBN 978-1-60558-794-3.

[12] R. S. Nikhil. Bluespec: A General-Purpose Approach to High-Level Synthesis Based on Parallel Atomic Transactions. In Philippe Coussy and Adam Morawiec, editor, *High-Level Synthesis - From Algorithm to Digital Circuit*, pages 129–146. Springer Netherlands, 2008.

[13] P. Sestoft. Deriving a lazy abstract machine. *J. Funct. Program.*, 7(3): 231–264, 1997.

[14] M. Sheeran. Hardware Design and Functional Programming: a Perfect Match. *Journal of Universal Computer Science*, 11(7):1135–1158, jul 2005.

[15] M. Sulzmann, M. M. T. Chakravarty, S. L. P. Jones, and K. Donnelly. System f with type equality coercions. In F. Pottier and G. C. Necula, editors, *TLDI*, pages 53–66. ACM, 2007. ISBN 1-59593-393-X.

[16] V. F. Turchin. The concept of a supercompiler. *ACM Trans. Program. Lang. Syst.*, 8(3):292–325, 1986.