Using Rewriting to Synthesize Functional Languages to Digital Circuits

Christiaan Baaij and Jan Kuper

Department of Electrical Engineering, Mathematics, and Computer Science, University of Twente, Postbus 217, 7500AE Enschede, The Netherlands
`{c.p.r.baaij,j.kuper}@utwente.nl`

Abstract. A straightforward synthesis from functional languages to digital circuits transforms variables to wires. The types of these variables determine the bit-width of the wires. Assigning a bit-width to polymorphic and function-type variables within this direct synthesis scheme is impossible. Using a term rewrite system, polymorphic and function-type binders can be completely eliminated from a circuit description, given only minor and reasonable restrictions on the input. The presented term rewrite system is used in the compiler for CλaSH: a polymorphic, higher-order, functional hardware description language.

1 Introduction

This paper describes the use of a Term Rewriting System (TRS) in the compiler for CλaSH [1, 2]. CλaSH is a polymorphic, higher-order, functional hardware description language. The purpose of the CλaSH compiler is to transform a description in a functional language to a format from which lithography machines can build an actual chip. The CλaSH compiler actually only provides a part of this transformation. It creates a low-level representation of the hardware, called a netlist; industry-standard tools are used for further processing. The translation from a (functional) description to a netlist is called *synthesis* in hardware literature. And the set of rules/transformations that together describe the conversion procedure from *description* to *netlist* is called a *synthesis scheme*.

The *synthesis scheme* used by the CλaSH compiler produces a specific *normal form* of the description. One aspect of this normal form is that the arguments and results of functions must have a representable type: a type whose values can be encoded by a fixed number of bits. This paper *only* describes the TRS that is used by the CλaSH compiler to eliminate, in a meaning-preserving manner, non-representable values from a functional description. Neither the exact normal form, the simplification TRS used to achieve this normal form, nor the complete synthesis scheme, are however presented. These aspects will be described in a future paper. This paper focuses on the TRS for non-representable value elimination, because it, among other things, transforms higher-order descriptions to first-order descriptions. Because first-order programs are susceptible to a greater range of analysis techniques [3], the described TRS has value in a broader context.

The next subsection gives both a definition for netlists, and an introduction to synthesis schemes by describing a specific instance for a small functional language. The definition and introduction are both informal, but hopefully instil an intuition for the process of transforming a functional description to actual hardware.

1.1 Netlists & Synthesis

A netlist is a textual description of a digital circuit [4]. It lists the components that a circuit contains, and the connections between these components. The connection points of a component are called ports, or pins. The ports are annotated with the bit-width of the values that flow through them. A netlist can either be hierarchical or flattened. In a hierarchical netlist, sub-netlists are abstracted to appear as single components, of which multiple instances can be created. By instantiating all of these instances, a flattened netlist can be created.

A synthesis scheme defines the procedure that transforms a (functional) description to a netlist. Synthesis schemes defined for different languages, which nonetheless have common aspects, will be called a synthesis scheme *family*. The CλaSH compiler uses a synthesis scheme, called $\mathcal{T}_{C\lambda}$, that is an instance of the larger synthesis scheme family that will be referred to as $\mathcal{T}$. The following aspects are shared by all instances of $\mathcal{T}$:

- It is completely syntax-directed.
- It creates a hierarchical netlist.
- Function *definitions* are translated to components, where the arguments of the function become the input ports, and the result is connected to the output port.
- Function *application* is translated to an instance of the component that represents the corresponding function. The applied arguments are connected to the input ports of the component instance.

To demonstrate $\mathcal{T}$, a simple functional language, $\mathcal{L}$, is introduced in Fig. 1. $\mathcal{L}$ is a pure, simply-typed, first-order functional language. A program in $\mathcal{L}$ consists of set of function definitions, which always includes the *main* function. Expressions in $\mathcal{L}$ can be: variable references, primitives, or function application. Fig. 3 shows a small example program defined in the presented functional language.

The synthesis scheme for $\mathcal{L}$, called $\mathcal{T}_{\mathcal{L}}$, is defined by two transformations: $[\![\ ]\!]_p$ and $[\![\ ]\!]_e$, in which $[\![\ ]\!]_p$ is initially applied to the *main* function to create the hierarchical netlist. A graphical, informal, definition of the $[\![\ ]\!]_p$ and $[\![\ ]\!]_e$

transformations is depicted in Fig. 2. Again, the purpose of this subsection is to give an intuition for the synthesis process, not to give a formal account of $\mathcal{T}_{\mathcal{L}}$. $[\![\ ]\!]_p$ creates a component definition for a function f, where input ports correspond to the argument of f. $[\![\ ]\!]_p$ also creates an output port for the result of the expression e, which is connected to the outcome of the $[\![\ ]\!]_e$ transformation applied to e.

Fig. 2 shows that $[\![\ ]\!]_e$ transforms an argument reference x to a connection with an input port x. Function application of a function f is transformed to a component instance of f. $[\![\ ]\!]_p$ will be called for the definition of f in case there is

$$
\begin{array}{lll}
p ::= & f\ \overline{x} = e;\ p & \text{Function definitions} \\
\quad\mid & \emptyset & \\
 & & \\
e ::= & x & \text{Argument reference} \\
\quad\mid & \otimes\ \overline{e} & \text{Primitive} \\
\quad\mid & f\ \overline{e} & \text{Function application}
\end{array}
$$

Fig. 1. $\mathcal{L}$: a simple functional language

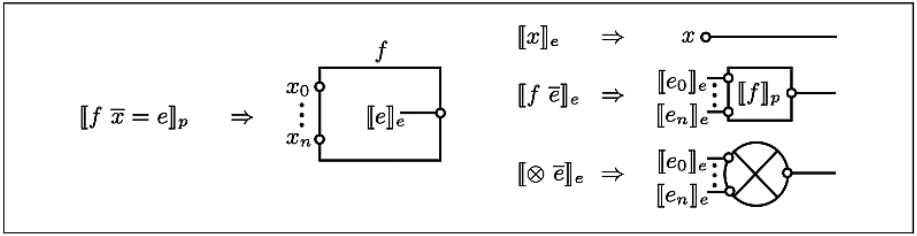


Fig. 2. $\mathcal{T}_{\mathcal{L}}$: A synthesis scheme for $\mathcal{L}$

$$
\begin{aligned}
&\mathit{double}\ x\ = x * x\\
&\mathit{main}\ x\ y = (\mathit{double}\ x) + (\mathit{double}\ y)
\end{aligned}
$$

Fig. 3. Example program in $\mathcal{L}$

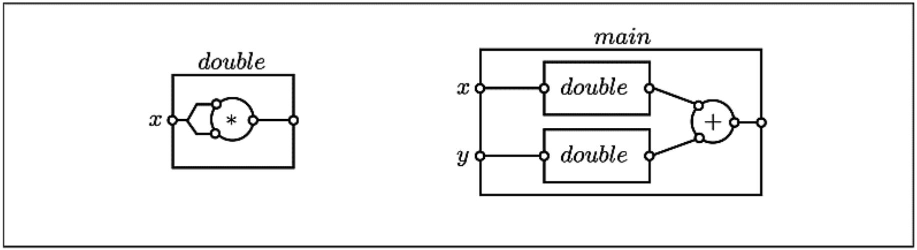


Fig. 4. Netlist of the example program in Fig. 3, created by $\mathcal{T}_{\mathcal{L}}$

no existing component definition. Arguments to f are recursively transformed by $[\![\ ]\!]_e$, and the outcome of these transformations are connected to the input ports of the component f. The process for the transformations of primitives is analogous to that of functions, except that $[\![\ ]\!]_p$ will not be called for the definitions.

Applying the synthesis scheme $\mathcal{T}_{\mathcal{L}}$ to the example program given in Fig. 3 results in the (graphical representation of the) netlists depicted in Fig. 4. The netlist representation of *main* shows that synthesis schemes belonging to $\mathcal{T}$ exploit the *implicit parallelism* present in (pure) functional languages: as there are no dependencies between the operands of the addition, they are instantiated side-by-side. During the actual operation of the circuit, electricity flows through all parts simultaneously, and the instances of *double* will actually be operating in parallel.

Synthesis of CλaSH CλaSH has a syntax and a semantics similar to the programming language Haskell [5] including some of language extensions of the Glasgow Haskell Compiler (GHC) [6]. These extensions include higher-rank polymorphism and existential datatypes. CλaSH and Haskell are so similar that every valid CλaSH description is also a valid Haskell program. Because CλaSH uses a synthesis transformation belonging to $\mathcal{T}$, called $\mathcal{T}_{C\lambda}$, the reverse relation does not hold. There are (many) Haskell programs that are not valid CλaSH descriptions. For example, recursive functions are not valid CλaSH descriptions: under $\mathcal{T}_{C\lambda}$, recursive application of a function f would lead to a recursive instance of the component f. Flattening the netlist would lead to infinitely many instantiations of the component f. Because such a netlist cannot be realized, the corresponding recursive function is currently deemed an invalid CλaSH description. Recursively defined (non-function) values are however allowed as they can be synthesized to feedback loops.

CλaSH uses an instance of the $\mathcal{T}$ family of synthesis schemes because it exploits the implicit parallelism of the functional descriptions, as shown earlier in Fig. 4. An important aspect of $\mathcal{T}$ is that the arguments and results of functions become the input and output ports of components. These ports are annotated with a bit-width so that it is known how many wires are needed to make connections between ports. Because CλaSH is a polymorphic, higher-order language, the arguments and results can however contain polymorphic or function-typed values. *It is generally impossible or impractical to represent such values by a fixed number of bits.*

In order to run $\mathcal{T}_{C\lambda}$, all values that cannot be represented by a fixed bit-width, will have to be eliminated from the functional description. The focus of this paper is a TRS that transforms the functional description in a meaning-preserving manner so that all non-representable values are eliminated. The presented TRS achieves its goal using both inlining and specialisation transformations [3].

The remainder of this paper is structured as follows: related work is described in the next section. CλaSH is desugared to a smaller Core language, and it is the Core language on which the TRS operates; Sect. 3 describes this Core language. Section 4 defines the (data)types which are considered non-representable, and the general process required for their meaning-preserving elimination. The rewrite rules of the TRS are described in Sect. 4.1. Properties of the TRS, including its non-termination, and the subsequent measures taken in the CλaSH compiler are discussed in Sect. 5. Conclusions are presented in Sect. 6.

2 Related Work

Functional Hardware Description Languages. SAFL [7] is a first-order hardware description language. As opposed to $\mathcal{T}_{C\lambda}$, which is used by CλaSH, SAFL uses a synthesis scheme that does not create a new component instance for every application of a function f. Instead, a component f is instantiated only once, and additional control and scheduling logic is inferred to safely approximate concurrent access.

BlueSpec SystemVerilog [8] is a hardware description language with a syntax similar to IEEE SystemVerilog standard. It has features also found in functional languages, such as higher-order functions and parametric polymorphism. The compilation from description to netlist is performed in two stages, which corresponds to the static and dynamic semantics of the language:

- A description is partially evaluated according to the static semantics, this includes the elimination/propagation of higher-order functions.
- The resulting description after partial evaluation is actually a set of rewrite rules. The second synthesis transformation instantiates all these rules in parallel, and adds scheduling logic in case there are conflicting preconditions [9].

So where the CλaSH compiler uses a TRS to eliminate non-representable values (such as those with a function type), the BlueSpec compiler uses a partial evaluator. There is however no account of the exact details of then partial evaluation mechanism in the Bluespec compiler, nor is there an exhaustive list of restrictions / requirements on the input programs.

Lava [10, 11] is a domain specific language *embedded in* Haskell. A hardware description in Lava is actually a Haskell program that uses combinators made available by the Lava library. These combinators wrap constructors of a graph datatype that represents a netlist. Synthesis of Lava descriptions is not performed in the traditional sense of transforming a description to a netlist. By *running* the Lava description, a Haskell program, the complete graph representing the netlist is simply calculated/constructed. Consequently, Lava gets the synthesis of higher-order, and recursive functions, *for free*: as long as the function *calculating* the graph terminates, a netlist can be created. Being an embedded language, Lava has disadvantages compared to a compiled language such as CλaSH:

- Because a program calculating the netlist graph cannot *observe* the (application of) individual functions, there can be no intuitive function-to-component mapping. As a result, only flattened netlists can be created.
- The rich set of *choice*-constructs in Haskell (also present in CλaSH), such as pattern-matching, cannot be reflected down to the netlists. Haskell's choice construct can be used to *guide the construction* of the netlist graph, but they cannot be *observed*. Consequently, a developer using Lava only has access to *choice*-functions offered by the Lava library.

Verity is a higher-order functional hardware description language with support for recursion (using a fix-point combinator) and mutable reference-cells. Verity uses a *semantics*-directed synthesis scheme called *Geometry of Synthesis (GoS)* [12]. GoS assumes a linear type system, that restricts the use of identifiers to *exactly once*. That means that arguments with a function type need to be instantiated only once, an aspect GoS exploits during synthesis. Given a higher-order function f, which has a function-type argument g, the component corresponding to f is given extra input and output ports. The extra output ports correspond to the input ports for g, and the extra input ports correspond to the output ports of g. When f is applied to a function h, an instance of both the f and h component are created, and the components are correctly connected to each other. CλaSH does not have a linear type-system, meaning an identifier with a function type can be applied multiple times. Because of this, the CλaSH compiler cannot use the synthesis approach for function-typed arguments as promoted by GoS.

Higher-Order removal methods. Reynolds-style defunctionalisation [13] is a well-known method for generating an equivalent first-order program from a higher-order program. Reynolds' method creates datatypes for arguments with a function-type. Instead of applying a higher-order function to a value with a function-type, it is applied to a constructor for the newly introduced datatype. Application of the functional argument is replaced by the application of a mini-interpreter. Given the following higher-order program:

```
uncurry f (a, b) = f a b
main x = (uncurry (+) x) + (uncurry (−) x)
```

Reynolds' method creates the following behaviourally equivalent first-order program:

```
data Function = Plus | Sub
apply Plus a b = (+) a b
apply Sub  a b = (−) a b

uncurry f (a, b) = apply f a b
main x = (uncurry Plus x) + (uncurry Min x)
```

Reynolds' method works on all programs, removes all functional arguments, and preserves sharing (a subject that will be discussed later). Although commonly defined and studied in the setting of the simply typed lambda calculus, there are also variants [14, 15] of Reynolds' methods that are correct within a polymorphic type system. The disadvantage of Reynolds' method is the introduction of the mini-interpreter (which takes on the form of the *apply* function in the example). Due to the parallel nature of $\mathcal{T}_{C\lambda}$, this interpreter and all of its corresponding functionality will be instantiated at the use sites of the interpreter. For the above example it means that the interpreter will be instantiated twice; and so will the included functionality: the adder and the subtracter. This method, in combination with $\mathcal{T}_{C\lambda}$, thus creates a lot of redundant hardware; it is this aspect that has precluded the use of Reynolds' method in the CλaSH compiler.

Many of the rewrite rules used by the TRS described in this paper can also be found in optimizing compilers for functional languages, such as GHC [16]. The rewrite rules presented by Peyton Jones and Santos [16] do however not guarantee a first-order normal form, which the TRS presented in this paper does (given certain restrictions on the input program).

Mitchell and Runciman [3] present a defunctionalisation method based on a TRS, which, like the TRS presented in this paper, also uses specialisation and inlining. The presented TRS can thus be seen as an extension to the work of Mitchell and Runciman:

- It provides transformations that additionally perform monomorphisation, which includes the specialisation of: higher-rank polymorphic arguments and existential datatypes.

- It can deal with recursive let-expressions.
- It works on a typed language, and uses this type information to determine when transformations should be applied.

3 Core Language

The syntactically rich CλaSH language is desugared to a smaller Core language, called Core$_{\text{HW}}$ (Fig. 5), by the CλaSH compiler. It is a Church-style polymorphic lambda-calculus extended with primitives, algebraic datatypes in combination with case-decomposition, and recursive let-bindings. Case-decompositions are either exhaustive in the constructors of the matched datatype, or include the default pattern. Recursive let-bindings are needed to define values/expressions that are self-referencing and are used to describe feedback loops. Fig. 5 gives the language definition of Core$_{\text{HW}}$, and uses, just like the rest of this paper, the notation described in Fig. 6.

CλaSH supports existential datatypes, and this aspect of the language is reflected in Core$_{\text{HW}}$. A data constructor K, for an existential datatype $\mathbf{T}\ \overline{\alpha}$, is first abstracted over the universally quantified type-variables $\overline{\alpha}$, followed by the existentially quantified type-variables $\overline{\beta}$. The type variables $\overline{\beta}$ brought into scope by a pattern in a case-decomposition correspond to the existentially-quantified type-variables of the datatype.

3.1 Synthesis of Core$_{\mathrm{HW}}$ using $\mathcal{T}_{C\lambda}$

The synthesis scheme $\mathcal{T}_{C\lambda}$ exploits all the implicit parallelism available in the Core$_{\mathrm{HW}}$ language. It does this by instantiating all expressions in a let-binding, and all alternatives of a case-decomposition, side-by-side (Fig. 7). $\mathcal{T}_{C\lambda}$ creates anchor points for let-binders so that variable references can be synthesized to connections to these anchor points.

Completely elaborating $\mathcal{T}_{C\lambda}$ falls outside of the scope of this paper. To at least convey an intuition for the synthesis performed by $\mathcal{T}_{C\lambda}$, an example program, and the corresponding netlist are shown in Fig. 8 and 9 respectively. The

Local variables: x, y, z
Global Variables: f, g
Type Variables: α, β

Data Constructor Types:
$K : \forall\overline{\alpha}.\forall\overline{\beta}.\overline{\tau} \rightarrow \mathbf{T}\ \overline{\alpha}$

Types:

$\tau, \sigma ::=$	
α	Variable reference
$\mid \tau \rightarrow \sigma$	Function Type
$\mid \mathbf{T}$	Datatype
$\mid \tau\ \sigma$	Type application
$\mid \forall\alpha.\sigma$	Polymorphic type

Expressions:

$e, u ::=$	
$x \mid f$	Variable reference
$\mid K \mid \otimes$	Data Constructor / Primitive
$\mid \Lambda\alpha.e \mid e\ \tau$	Type abstraction / application
$\mid \lambda x : \sigma.e \mid e\ u$	Term abstraction / application
$\mid \mathbf{let}\ \overline{x : \sigma = e}\ \mathbf{in}\ u$	Recursive let-binding
$\mid \mathbf{case}\ e\ \mathbf{of}\ \overline{p \rightarrow u}$	Case-decomposition

Patterns:

$p ::= \_$	Default case
$\mid\ K\ \overline{\beta}\ \overline{x : \sigma}$	Match data constructor

Fig. 5. The Core$_{\text{HW}}$ calculus

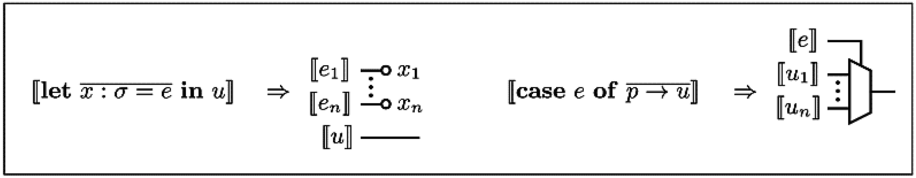
$$\begin{array}{llll}
\mathbf{T}\,\overline{\sigma} & \equiv \mathbf{T}\,\sigma_1\,\ldots\,\sigma_n & e\,\overline{u} & \equiv e\,u_1\,\ldots\,u_n \\
\overline{\tau} \to \sigma & \equiv \tau_1 \to \ldots \to \tau_n \to \sigma & \lambda\overline{x:\sigma}.e & \equiv \lambda x_1 : \sigma_1.\;\ldots\;\lambda x_n : \sigma_n.e \\
\forall\overline{\alpha}.\sigma & \equiv \forall\alpha_1.\;\ldots\;\forall\alpha_n.\sigma & \overline{x:\sigma=e} & \equiv \{x_1 : \sigma_1 = e_1,\;\ldots\;,\;x_n : \sigma_n = e_n\} \\
 & & \overline{p \to u} & \equiv \{p_1 \to u_1,\;\ldots\;,\;p_n \to u_n\}
\end{array}$$

$$K\,\overline{\beta}\,\overline{x:\sigma} \;\equiv\; K\,\beta_1\,\ldots\,\beta_n\,(x_1 : \sigma_1)\,\ldots\,(x_m : \sigma_m)$$

Fig. 6. Notation

simultaneous presence of all alternatives in a case-decomposition, and all binders in a let-binding, has consequences for the *sharing* behaviour of expressions.

Sharing is normally defined as the *re-use of the result of a computation* by other expressions. In a digital circuit, sharing means connecting the output port of one component to the input ports of multiple other components. This aspect can be observed in Fig. 9, where the result of the multiplication is shared by the addition and the subtraction. Results that can be shared, instead of recomputed, will reduce the total size of the circuit. The rewrite rules of the TRS should thus take the effects of sharing under $\mathcal{T}_{C\lambda}$ into account, as any loss in sharing increases the size of the circuit.



$[\![\textbf{let } \overline{x : \sigma = e} \textbf{ in } u]\!] \Rightarrow$
$[\![e_1]\!]$ x_1
$[\![e_n]\!]$ x_n
$[\![u]\!]$
$[\![\textbf{case } e \textbf{ of } \overline{p \to u}]\!] \Rightarrow$
$[\![e]\!]$
$[\![u_1]\!]$
$[\![u_n]\!]$

Fig. 7. Synthesis of **let** and **case**

$\lambda x : \mathit{Bool}.\lambda y : \mathit{Int} \circ \textbf{let}$
$\quad z : \mathit{Int} = y * y$
$\textbf{in case } x \textbf{ of}$
$\quad \mathit{True} \;\; \to z + 1$
$\quad \mathit{False} \;\; \to z + 1$

Fig. 8. Example program using **let** and **case**

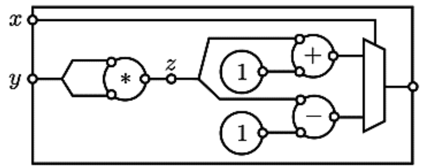


Fig. 9. Netlist of the example program in Fig. 8, created by $\mathcal{T}_{C\lambda}$

4 Eliminating Non-representable Types

$\mathcal{T}_{C\lambda}$ can only synthesize functional descriptions if arguments and results of expressions can be given a fixed bit-encoding. There are straightforward encodings for certain primitive datatypes, and certain algebraic datatypes. Datatypes with a fixed bit-encoding are called *representable*. Deriving a fixed bit-encoding for the following types is either not desired, or not possible:

– Function types
– (Higher-rank) polymorphic types
– Recursively defined datatypes.

– Datatypes that are composed of types that are not representable.

This section shows the TRS that eliminates non-representable values from the function hierarchy. It eliminates such values completely given that the input adheres to the following restriction:

– That both the arguments and the result of the *main* function, and the arguments and result of the used primitives, are representable.

The TRS uses a combination of inlining and specialisation, where specialisation takes on two forms:

– Specialisation of a function on one of its arguments.
– Elimination of a case-decomposition based on a known constructor.

The rewrite rules in this paper are presented using the format depicted in Fig. 10. In all of these rewrite rules, the expression above the horizontal bar is the expression that has to be matched before performing the rewrite rule, and the expression below the horizontal bar is the result after applying the rewrite rule.

NAME OF THE REWRITE RULE

$$\frac{\text{Matched Expression}}{\text{Resulting Expresson}} \qquad \begin{array}{l}\langle\text{Additional Preconditions}\rangle \\ \langle\text{Additional Definitions}\rangle \\ \langle\text{Updated Environment}\rangle\end{array}$$

Fig. 10. Format for Rewrite Rules

Some rewrite rules have additional preconditions, and the rewrite is only applied when these preconditions hold. Other rewrite rules have additional definitions which they use in the resulting expressions. All rewrite rules always have access to the global environment, Γ, which holds all top-level binders. There are some rewrite rules that create new top-level binders, and therefore update the global environment.

The rewrite rules have access to the following functions:

FV e	Calculates the free variables; works for types and terms.
$e\ [x := u]$	A capture-free substitution of a variable reference x, by the expression or type u, in the expression e.
$\Gamma@f$	The expression belonging to a global binder f in the

environment Γ.

NONREP τ Determines if τ is a non-representable type.

Before the TRS starts, all variables are made unique, and all variable references are updated accordingly. Any new variables introduced by the rewrite rules will be unique by construction. Having hygienic expressions prevents accidental free-variable capture, and makes it easier to define meaning-preserving rewrite rules.

4.1 Rewrite rules

The first three rewrite rules, τ-REDUCTION, LETTYAPP, and CASETYAPP, propagate type information downwards into an expression. By either removing type-variables, propagating type-information to a location for specialisation, or propagating type information to a primitive or constructor, these rewrite rules aid in the elimination of polymorphism.

τ-REDUCTION
$$\frac{(\Lambda\alpha.e)\ \tau}{e\ [\alpha := \tau]}$$

LetTyApp
$$\frac{(\textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ u)\ \tau}{\textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ (u\ \tau)}$$

CaseTyApp
$$\frac{(\textbf{case}\ e\ \textbf{of}\ \overline{p \to u})\ \tau}{\textbf{case}\ e\ \textbf{of}\ \overline{p \to (u\ \tau)}}$$

The next three rewrite rules, LamApp, LetApp, and CaseApp, propagate values, including non-representable ones, downwards into the expression. LamApp is preferred over β-reduction to preserve sharing. CaseApp creates a let-binding, instead of propagating the applied expression towards all alternatives, to preserve sharing. The next rewrite rule, LiftNonRep, removes let-binders introduced by LamApp and CaseApp in case they bind non-representable values.

LamApp
$$\frac{(\lambda x : \sigma . e)\ u}{\textbf{let}\ \{x = u\}\ \textbf{in}\ e}$$

LetApp

$$\frac{(\textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ u)\ e_0}{\textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ (u\ e_0)}$$

CaseApp

$$\frac{(\textbf{case}\ e\ \textbf{of}\ \overline{p \rightarrow u})\ u_0}{\textbf{let}\ \{x = u_0\}\ \textbf{in}\ (\textbf{case}\ e\ \textbf{of}\ \overline{p \rightarrow (u\ x)})}$$

LiftNonRep removes a let-binder, $x_i : \sigma_i = e_i$ (with a non-representable type σ_i), and substitutes references to x_i in the rest of the let-binding with an (application of a) variable reference to a *new*, *global*, binder: f. The new global binder, f, binds the original expression e_i which is abstracted over the free local (type) variables of e_i; all references to x_i are substituted with an (application of a) variable reference to f. The LiftNonRep rewrite rule uses the $\cup_\alpha$ operator to indicate that the global environment is only updated with the new binder, f, if an α-equivalent expression is not already present. In case an α-equivalent expression is present in the environment, the transformed expression will refer to that existing global binder instead.

LiftNonRep

$$\frac{\textbf{let } \{b_1; ...; b_{i-1}; x_i : \sigma_i = e_i; b_{i+1}; ...; b_n\} \textbf{ in } u}{(\textbf{let } \{b_1; ...; b_{i-1}; b_{i+1}; ...; b_n\} \textbf{ in } u)\ [x := f\ \overline{\alpha}\ \overline{z}]} \quad \text{Preconditions: NONREP}(\sigma_i)$$

$$\text{Definitions: } (\overline{\alpha}, \overline{y}) = \text{FV}(e_i); \quad \overline{z} = \overline{y} - \{x_i\}$$

$$\text{New Environment: } \Gamma \cup_\alpha \{(f, \Lambda\overline{\alpha}.\lambda\overline{z}.e_i[x_i := f\ \overline{\alpha}\ \overline{z}])\}$$

The previous rewrite rules either propagated non-representable values downwards into the expression, or lifted those values out of the expression. The next two sets of rewrite rules remove non-representable values by specialisation. The TypeSpec and NonRepSpec provide function argument specialisation. CaseLet, CaseCase, InlineNonRep, and CaseCon, together achieve specialisation by eliminating case-decompositions of known constructors (of non-representable datatypes).

The TypeSpec rewrite rule matches on a type application of a variable reference to a global binder, f. The application is replaced by a reference to the *new* global binder f'. The new binder f' is defined in terms of the body of f specialized on the type τ. NonRepSpec behaves similarly to TypeSpec for the application on non-representable arguments. The difference is that the expression of the new binder, f', is abstracted over the free variables of the specialised argument; the transformed expression also takes these free variables into account.

Both TYPESPEC and NONREPSPEC use the $\cup_\alpha$ operator to indicate that the global environment is only updated with a new binder if an α-equivalent specialization is not already present. In case an α-equivalent specialisation is present in the environment, the transformed expression will refer to that existing global binder instead.

$$\textsc{TypeSpec} \qquad \frac{(f\ \overline{e})\ \tau}{f'\ \overline{e}} \qquad \text{Preconditions: } \mathrm{FV}(\tau) \equiv \emptyset$$
$$\text{New Environment: } \mathit{\Gamma} \cup_\alpha \{(f', \lambda\overline{x}.(\mathit{\Gamma}@f)\ \overline{x}\ \tau)\}$$

$$\textsc{NonRepSpec} \quad \frac{(f\ \overline{e})\ (u\!:\!\sigma)}{f'\ \overline{e}\ \overline{y}} \quad \begin{array}{ll}\text{Preconditions: } & \mathrm{NONREP}(\sigma) \wedge \mathrm{FV}(\sigma) \equiv \emptyset \\ \text{Definitions: } & \overline{y} = \mathrm{FV}(u)\end{array}$$
$$\text{New Environment: } \mathit{\Gamma} \cup_\alpha \{(f', \lambda\overline{x}.\lambda\overline{y}.(\mathit{\Gamma}@f)\ \overline{x}\ u)\}$$

The CASELET is required in specialising expressions that have a non-representable datatype. Taking the let-binders out of the case-decomposition does not affect the sharing behaviour so can be applied blindly. There is no free variable capture in the alternatives because all variables are made unique before running the TRS.

The CaseCase rewrite rule is only applied if the subject of a case-decomposition has a non-representable datatype. CaseCase is not applied blindly because the alternatives in a case-decomposition are evaluated in parallel in the eventual circuit. So the CaseCase rewrite rule generates a larger number of alternatives than present in the matched expression. A larger number of alternatives results in a larger circuit. Even though CaseCase makes the circuit larger, the intention of CaseCase is to eventually expose the constructor of the non-representable datatype to CaseCon. CaseCon eliminates the case-decomposition, and subsequently amortizes the increase in circuit size induced by CaseCase.

InlineNonRep is only applied if the subject of a case expression is of a non-representable datatype, as inlining breaks down the component hierarchy. All bound variables in the inlined expression are regenerated, and variable references updated accordingly. This preserves the assumptions made by the other rewrite rules that all variables are unique.

The CaseCon rule comes in two variants:

- A case-decomposition with a constructor application as the subject, and a matching constructor pattern.

- A case-decomposition with a constructor application as the subject, with *no* matching constructor pattern.

CASECON only creates a let-binding if the constructor in the subject exactly matches the constructor of an alternative. When the default pattern is matched, the case-decomposition is simply replaced by the expression belonging to the default alternative. Case-decompositions in Core$_{\text{HW}}$ are exhaustive, either by enumerating all the constructors, or by including the default pattern. This means that when a constructor applications is the subject of a case-decomposition, CASECON will always remove that case-decomposition.

CASELET

$$\frac{\textbf{case}\ (\textbf{let}\ \overline{x:\sigma = e}\ \textbf{in}\ e_1)\ \textbf{of}\ \overline{p \rightarrow u}}{\textbf{let}\ \overline{x:\sigma = e}\ \textbf{in}\ (\textbf{case}\ e_1\ \textbf{of}\ \overline{p \rightarrow u})}$$

CASECASE Preconditions: NONREP(σ)

$$\frac{\textbf{case}\ (\textbf{case}\ e\ \textbf{of}\ \{p_1 \rightarrow u_1;\ \ldots\ ;\ p_n \rightarrow u_n\} : \sigma)\ \textbf{of}\ \overline{p \rightarrow u}}{\textbf{case}\ e\ \textbf{of}\{p_1 \rightarrow \textbf{case}\ u_1\ \textbf{of}\ \overline{p \rightarrow u};\ \ldots\ ;\ p_n \rightarrow \textbf{case}\ u_n\ \textbf{of}\ \overline{p \rightarrow u}\}}$$

InlineNonRep — Preconditions: NONREP(σ)

$$\frac{\textbf{case}\ (f\ \overline{e}) : \sigma\ \textbf{of}\ \overline{p \rightarrow u}}{\textbf{case}\ ((\Gamma @ f)\ \overline{e})\ \textbf{of}\ \overline{p \rightarrow u}}$$

CaseCon

$$\frac{\textbf{case}\ K_i\ \overline{\tau_\forall}\ \overline{\tau_\exists}\ \overline{e}\ \textbf{of}\ \{...; K_i\ \overline{\beta}\ \overline{x : \sigma} \rightarrow u_i; ...\}}{(\textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ u_i)\ [\overline{\beta} := \overline{\tau_\exists}]}$$

$$\frac{\textbf{case}\ K_i\ \overline{\tau_\forall}\ \overline{\tau_\exists}\ \overline{e}\ \textbf{of}\ \{\overline{p_{j \neq i} \rightarrow u}; \_ \rightarrow u_0\}}{u_0}$$

5 Discussion

5.1 Completeness

The first set of rewrite rules (τ-Reduction - LiftNonRep) propagates or removes non-representable values for those syntactical elements on which the specialisation rewrite rules do not match. The second set of rewrite rules (TypeSpec - CaseCon) remove the non-representable values through specialisation. All rewrite rules together hence remove all non-representable values from the function hierarchy (given the restrictions in Section 4).

The restrictions on primitives are needed because those cannot be specialized on their argument, nor can their definitions be inlined. The restriction that the result type of *main* cannot be a non-representable datatype, ensures that any expression calculating a non-representable datatype is either:

- the subject of a case-decomposition, which will be removed by the TRS,
- or, unreachable, and can be removed by dead-code elimination.

The CλaSH compiler applies the transformations in a specific order: a traversal with TypeSpec, followed by a traversal with NonRepSpec, are applied *after* all the other transformations have been applied exhaustively. Neither the correctness of the individual transformations, nor the guarantee of a first-order normal form, are dependent on this specific ordering of transformations. The argument-specialisation rewrite rules are applied last, so that the fewest number of new functions is introduced, and the original function hierarchy is preserved as much as possible. Because TypeSpec and NonRepSpec do not create expressions on which the other rewrite rules match, all rewrite rules have been applied exhaustively after the traversals with TypeSpec and NonRepSpec.

5.2 Termination

All rewrite rules are exhaustively applied during a (repeated) bottom-up traversal of an expression. InlineNonRep has to be applied using a bottom-up traversal, as a top-down traversal could lead to non-termination when inlining a recursive function. Using a bottom-up traversal for TypeSpec and NonRepSpec introduces the fewest number of lambda-abstraction in the specialized expressions.

There are several (combinations of) rewrite rules that induce non-termination of the unconstrained TRS. The CλaSH compiler has heuristics in place that constrain the application of certain rewrite rules to ensure termination. When one of the termination measures is triggered, non-representable values remain present in the description. $\mathcal{T}_{C\lambda}$ will not be able to transform the description to a netlist when that happens.

It should be noted that these termination measures are only trigged on functions that contain (mutually) recursive function calls, or have a (mutually) recursive datatype as a result; functions which cannot be synthesized by $\mathcal{T}_{C\lambda}$ anyway. It can hence be said that the unconstrained TRS terminates for all *usefull* programs.

InlineNonRep restriction The precondition of InlineNonRep already limits the locations where inlining is applied, exhaustive application of this rewrite rule can however still induce non-termination when dealing with recursive functions.

Additionally, although the TRS does not contain β-reduction as one of the rewrite rules, LAMAPP, LIFTNONREP, INLINENONREP, and CASECON together behave like β-reduction. This means that the typed version of $(\lambda x \rightarrow x\ x)\ (\lambda x \rightarrow x\ x)$:

$$\begin{array}{l}\textbf{data}\ T = C\ (T \rightarrow \mathit{Int}) \\ (\lambda x \rightarrow \textbf{case}\ x\ \textbf{of}\ C\ h \rightarrow h\ x)\ (C\ (\lambda x \rightarrow \textbf{case}\ x\ \textbf{of}\ C\ h \rightarrow h\ x))\end{array}$$

induces non-termination. To prevent either situation from happening, a function f can only be inlined *once* at all use sites within a function g, for every pair of f and g.

NonRepSpec restriction Specialization performed by NONREPSPEC can induce non-termination when a recursive function f has an argument that accumulates non-representable values. To ensure termination, a NONREPSPEC is only applied to a function m number of times, where m can be set by the user of the CλaSH compiler.

6 Conclusions

The CλaSH compiler uses a synthesis scheme, $\mathcal{T}_{C\lambda}$, that produces a description that has specific normal from. One aspect of this normal form is that arguments and results of expressions have types for which a fixed bit-encoding exists. For

$\mathcal{T}_{C\lambda}$, non-representable values are those values for which no fixed bit-encoding can be determined. The TRS presented in this paper removes all non-representable values from a function hierarchy while preserving the behaviour, given only minor restrictions on this function hierarchy. These restrictions are: that neither the *main* function nor the primitives of CλaSH, can have arguments or results of a non-representable type. These restrictions do however not limit the use polymorphism or higher-order functionality in the rest of the description. We, the authors of this paper, deem these restrictions reasonable for the application domain of CλaSH: creating digital circuits.

Although the CλaSH compiler cannot synthesize recursive function, this limitation is (slightly) amortized by a set of primitives that capture certain recursive patterns. Such functions / primitives include the *map* and *foldl* functions for fixed-length vectors. Using custom rules for these primitives, the CλaSH compiler can correctly synthesize the residual higher-order functionality that is left after normalization. A restriction that still holds for the use of these primitives is that they should *not* have a non-representable result.

Future Work The complete $\mathcal{T}_{C\lambda}$ synthesis scheme, the normal-form of the Core_{HW} language which $\mathcal{T}_{C\lambda}$ produces, and the simplification TRS of the CλaSH compiler will be described in a future paper. To reduce the number of traversals needed to reach the first-order normal from, the strategy of the presented TRS and its implementation within the CλaSH compiler are also subject to further investigation. Aside from improving the TRS, we are also extending the CλaSH compiler to support the synthesis of recursive functions that can be unrolled at compile-time.

References

1. Baaij, C.P.R., Kooijman, M., Kuper, J., Boeijink, W.A., Gerards, M.E.T.: CλaSH: Structural Descriptions of Synchronous Hardware using Haskell. In: Proceedings of the 13th Conference on Digital System Design, USA, IEEE Computer Society (September 2010) 714–721
2. Gerards, M.E.T., Baaij, C.P.R., Kuper, J., Kooijman, M.: Higher-Order Abstraction in Hardware Descriptions with CλaSH. In: Proceedings of the 14th Conference on Digital System Design, USA, IEEE Computer Society (August 2011) 495–502
3. Mitchell, N., Runciman, C.: Losing Functions without Gaining Data. In: Proceedings of the second Symposium on Haskell, ACM (September 2009) 13–24
4. Frankau, S.: Hardware Synthesis from a Stream-Processing Functional Language. PhD thesis, University of Cambridge (July 2004)
5. Peyton Jones, S., ed.: Haskell 98 Language and Libraries. Volume 13 of Journal of Functional Programming. (2003)
6. The GHC Team: The GHC Compiler, version 7.6.1. `http://haskell.org/ghc` (January 2013)

7. Mycroft, A., Sharp, R.: A Statically Allocated Parallel Functional Language. In: Proceedings of the 27th International Colloquium on Automata, Languages and Programming, Springer-Verlag (2000) 37–48
8. Nikhil, R.S.: Bluespec: A General-Purpose Approach to High-Level Synthesis Based on Parallel Atomic Transactions. In Philippe Coussy and Adam Morawiec, ed.: High-Level Synthesis - From Algorithm to Digital Circuit. Springer Netherlands (2008) 129–146
9. Hoe, J.C., Arvind: Hardware Synthesis from Term Rewriting Systems. In: Proceedings of the tenth International Conference on VLSI. (1999) 595–619
10. Bjesse, P., Claessen, K., Sheeran, M., Singh, S.: Lava: Hardware Design in Haskell. In: Proceedings of the third International Conference on Functional Programming (ICFP), ACM (1998) 174–184
11. Gill, A.: Type-Safe Observable Sharing in Haskell. In: Proceedings of the second Haskell Symposium, ACM (Sep 2009) 117–128
12. Ghica, D.R.: Geometry of Synthesis: A structured approach to VLSI design. In: Proceedings of the 34th annual Symposium on Principles of Programming Languages (POPL), ACM (2007) 363–375
13. Reynolds, J.C.: Definitional Interpreters for Higher-Order Programming Languages. In: Proceedings of the 25'th ACM National Conference, ACM Press (1972) 717 – 740
14. Pottier, F., Gauthier, N.: Polymorphic Typed Defunctionalization. In: Proceedings of the 31st Symposium on Principles of Programming Languages (POPL), ACM (2004) 89–98

15. Bell, J.M., Bellegarde, F., Hook, J.: Type-Driven Defunctionalization. In: Proceedings of the second International Conference on Functional Programming (ICFP). (1997) 25–37
16. Peyton Jones, S., Santos, A.: Compilation by Transformation in the Glasgow Haskell Compiler. In: Functional Programming Workshops in Computing, Springer-Verlag (1994) 184–204