Shallowly Embedded, Quantitatively Typed DSL for Synchronous System Design

PHD FORUM SUBMISSION

Rui Chen
School of Electrical Engineering and Computer Science
KTH Royal Institute of Technology
Stockholm, Sweden
ruich@kth.se

***Abstract*—Electronic system level (ESL) design requires high-level language(s) to provide a unified viewpoint of heterogeneous systems. To achieve this goal, we may employ a family of embedded domain-specific languages (EDSLs) linked by the same host language. As the first step of this idea, this extended abstract presents ongoing research on an EDSL for synchronous system design, whose terms are constrained by the quantitative type theory and can be easily mapped to their software/hardware implementations.**

***Index Terms*—synchronous systems, embedded domain-specific languages, hardware/software co-design, quantitative type theory**

I. Introduction

An electronic system level (ESL) design process desires a unified high-level language to formally specify and model heterogeneous systems, ideally as executable models with clear correspondence to their implementations [1]. In this way, a designer's understanding of systems and their integration is improved; meanwhile, successive implementation and verification steps in the design process can be better integrated. However, designing or introducing such a language is challenging because each sub-system of a heterogeneous system may have its behaviour model and desire a unique paradigm for specification.

Introducing a family of *embedded domain-specific languages* (EDSLs) that are embedded in the same general-purpose programming language (the host language) is promising to address the challenge of heterogeneous system modelling and specification. Each DSL can have tailored notations and constructors promoted for a specific kind of sub-system. By embedding these DSLs into the same host language, the integration of models of sub-systems is also guaranteed.

As the first step of defining a family of EDSLs for system design, we present the design and implementation of an EDSL for synchronous system design in this research. The EDSL is shallowly embedded in the functional programming language Idris2 and consequently inherits its quantitative type system that combines the dependent and linear type systems. With the shallow embedding, terms of the DSL are represented by terms of the host language. Hence, DSL terms can be easily integrated with other models specified in the same language. Meanwhile, utilising the fine-grained quantitative type system allows us to specify terms and their compositions in the DSL precisely so that each term directly corresponds to an implementation of a synchronous system.

The proposed DSL is implemented with the *tagless-final approach* [2] in which interfaces are utilised to specify the DSL's syntax and *any* implementation of these interfaces forms a semantics of the DSL. Specifically, the DSL is defined by two sets of interfaces corresponding to stateless (compositional) and stateful (sequential) behaviour specification. Further, the DSL is implemented with two semantics so that terms in the defined DSL can be interpreted as either terms in the host language (Idris2) or register transfer level (RTL) designs specified in the Verilog HDL. This implementation allows us to freely map a DSL term's sub-terms to their software or hardware implementations. Hence, hardware/software co-design can be conducted with the proposed DSL.

As an ongoing research, we summarise the *planned* contributions of this research as follows:

- we propose a quantitatively typed DSL for synchronous system specification and modelling;
- we demonstrate a shallow embedding of the proposed DSL in Idris2 as its implementation, which is ready to be used for hardware/software co-design; and
- we demonstrate that another EDSL with the same implementation methodology may be transformed into the proposed EDSL via functions in the host language, which confirms that a family of EDSLs can be employed to carry an ESL design process.

Among these contributions, the following has been *achieved* at the time when this extended abstract is written:

- the syntax of the proposed DSL defined as interfaces in Idris2;
- the implementation of the stateless/compositional behaviour specification part of the DSL (the compositional layer) as Idris2 terms and HDL code generator; and

This research was partially funded by the Sweden's Innovation Agency (Vinnova) via the NFFP7 project 2019-02743 TRANSFORM - Design transformation for correct-by-construction design methodology, and the Advanced and innovative digitalization project 2021-02484 EARLY BIRD – Seamless System Design from Concept Phase to Implementation.

```
interface Compositional crepr where
  lam: {a: _} -> {bs: _} -> {b: _} -> {auto prf1: Sig a}
    -> {auto prf2: All Sig bs} -> {auto prf3: Sig b}
    -> (crepr (HList []) a -> crepr (HList bs) b)
    -> crepr (HList $ [a]++bs) b
  app:... -> crepr (HList $ [a]++bs) b -> crepr (HList as) a
    -> crepr (HList $ as++bs) b
  prod:... -> crepr (HList as) a -> crepr (HList bs) b
    -> crepr (HList $ as++bs) (a, b)
  fst:... -> crepr (HList $ as) (a, b) -> crepr (HList as) a
  snd:... -> crepr (HList $ as) (a, b) -> crepr (HList as) b
```

(a) The definition of stateless glues for composing terms.

```
interface Primitive crepr where
  const: {len:_} -> BitVec len
      -> crepr (HList []) (BitVec len)
  add: {len:_} -> crepr (HList []) (BitVec len)
    -> crepr (HList []) (BitVec len)
    -> crepr (HList []) (BitVec $ S len)
```

(b) The primitives/atomic components defined by the target platform.

Listing 1: The definition of the compositional layer.

- the implementation of the stateful/sequential behaviour specification part of the DSL (the sequential layer) as Idris2 terms and a partial implementation of the corresponding HDL code generator.

II. Compositional Layer

The definition of the compositional layer of the proposed DSL is shown in Listing 1. In this definition, a system to be specified is of the type `crepr (HList as) a`, which specifies that the system has a possibly empty list of inputs of type `HList as` and an output of type `a`. This definition may be understood as either a component-based design framework [3] with glues defined in Listing 1a and components defined in Listing 1b or a *subset* of lambda calculus (Listing 1a) with constants (Listing 1b). By subset, we mean that the abstraction rule (`lam`) is restricted by the predicate `Sig: Type -> Type` so that only bit-vectors (of the type `BitVec`) and their product can be abstracted. With this restriction, there is no way to define higher-order functions within the DSL. Consequently, only systems with a finite number of computations may be specified in and generated from this DSL. On the other hand, we can still utilise functions in the host language (macros of the DSL) to generate arbitrarily large systems.

Terms in the compositional layer are interpreted as terms in Idris2 by specialising the constructor `repr: Type -> Type` with the following type:

```
record EvalC a b where
  constructor MkEvalC
  evalC: (a -> b)
```

Implementing the interface with the `EvalComp` type is straightforward in Idris2 because, in essence, we are implementing pure functional language in a pure functional language. Its implementation as an *HDL code generator* relies on the following type:

```
record HDLComp a b where
  constructor MkHDLC
  genHDLC: State Nat (Comp a)
```

Even though this type consists of a state monad, the state is employed to generate unique names for HDL constructors only. The major part of the implementation of the HDL generator is still context-free, which confirms that terms in the compositional layer directly correspond to their implementations.

```
interface Compositional crepr
       => Sequential (crepr : Type -> Type -> Type)
           (repr: Type -> Type -> Type -> Type) | repr where
  llam:... -> (crepr (HList []) a -> repr s (HList bs) b)
     -@ repr s (HList $ a::bs) b
  app1:... -> (1 _: repr s (HList $ a::bs) b)
     -> crepr (HList as) a -> repr s (HList $ as++bs) b
  app2:... -> crepr (HList $ a::bs) b -> repr s (HList as) a
     -@ repr s (HList $ as++bs) b
  app3:... -> repr s2 (HList $ a::bs) b
     -@ repr s1 (HList as) a -@ repr (s1, s2) (HList $
     ↪  as++bs) b
```

(a) The definition of composition rules of the sequential layer.

```
interface Register
          (repr: Type -> Type -> Type -> Type) where
  constructor MkReg
  1 get:... -> repr a (HList []) a
  1 set:... -> repr a (HList []) (a, b)
     -@ repr a (HList []) b
```

(b) The definition of registers.

Listing 2: The definition of the sequential layer.

III. Sequential Layer

The definition of the sequential layer is shown in Listing 2 in which a system with its state of type `s`, list of inputs of type `HList as` and output of type `b` is represented by the type `repr s (HList as) b`. This definition employs the *linearity* enabled by the quantitative type theory to specify that *the number of states in a system is an invariant during its construction*, which allows us to specify limited resources at the specification and modelling stage of a design process.

Terms in the sequential layer are interpreted in Idris2 as Kleisli arrows of state monads whose state is a resource modelled by the linearity, i.e. as the following types:

```
data Sys: Type -> Type -> Type -> Type where
  MkSys: (1 _: a -> LState (!* s) b) -> Sys s a b
```

By unwrapping the state monad above `LState (!* s) b` to `(1 _: !* s) -> LPair (!* s) b`, it is clear that terms of the `Sys` type are all *Mealy machines*. The sequential layer's implementation as an HDL code generator is very close to the compositional layer's implementation. The only difference is that target terms will be of the type

```
record Seq a where
  constructor MkSeq
  1 reg: RegAssign
  comp: Comp a
```

in which the usage of registers is restricted.

IV. Future Works

As the next step of the research, we plan to finish the implementation of the sequential layer and demonstrate that a term that cannot be specified in the DSL, e.g. the application of a parallel skeleton, may be transformed into this DSL by methods proposed in [2].

References

[1] G. Martin, B. Bailey, and A. Piziali, *ESL design and verification: a prescription for electronic system level methodology*. Elsevier, 2010.

[2] J. Carette, O. Kiselyov, and C.-c. Shan, "Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages," *Journal of Functional Programming*, vol. 19, no. 5, pp. 509–543, 2009.

[3] J. Sifakis, "System design automation: Challenges and limitations," *Proceedings of the IEEE*, vol. 103, no. 11, pp. 2093–2103, 2015.