Master thesis project proposal

Haskell library for teaching the basics of digital design with the help of rich types, with emphasis on designing for and running on FPGAs.

Matthías Páll Gissurarson
pallm@student.chalmers.se

Suggested Supervisor at CSE: Mary Sheeran

Relevant completed courses:

TDA452 - Functional programming
TDA342 - Advanced functional programming
DAT060 - Logic in computer science
DAT280 - Parallel functional programming
DAT151 - Programming language technology
TDA293 - Software engineering using formal methods
DAT140 - Types for programs and proofs

March 23, 2017

Introduction

Hardware design is intricate and complex, and often difficult for programmers to get into. Descriptions of algorithms for hardware are more low-level than programmers are used to, and depend on other properties such as timing, how much space/area the circuit occupies, its depth and power usage. Reasoning about these algorithms and hardware descriptions has usually been delegated to experts which focus on the hardware design, while programmers view the hardware mostly as a black box and care only for specific details on a need-to-know basis.

Recently, Field-Programmable Gate Arrays (FPGAs) have been growing in popularity. An FPGA is an integrated circuit that can be configured and re-programmed after it has been manufactured (hence "field-programmable") [1]. FPGAs provide the capability for programmers to create specially designed circuits for a wide class of problems, which can accelerate their programs by a non-trivial degree. The recent announcement of servers in the cloud with attached FPGAs [2] [3] hint that we are at a turning-point for the availability of FPGAs to the average programmer or scientist. This paradigm shift presents a challenge to the status quo, causing programmers and scientists to re-evaluate their understanding of hardware, and to stop viewing hardware only as a black box, and evaluate whether custom circuits could be applied to their problem domain. To assist with this evaluation and help with the transition to FPGA focused programming we propose to create a Haskell library for teaching the basics of digital design. Haskell is a functional programming language that has been growing in popularity in the recent decade, and has a wide user base. Libraries for creating hardware-oriented descriptions of algorithms already exist for Haskell [4] [5] [6], and would be used as a base for the library. We would use the rich type system in Haskell to provide descriptive types for the library, which users would then use to gain a deeper understanding into the specifics of digital design. This would entail using new features of the type system in Haskell such as the `TypeInType` extension [7] and SMT solvers [8] to express non-functional properties of the hardware-oriented descriptions in the type system and solve constraints thereof, in a similar manner to how Nils Danielsson expressed algorithmic complexity for lazy data-structures [9]. This will allow for the user to lean on the type system to catch mistakes in the design and aid in the understanding of digital design.

Context

Using dependent types to assist with hardware-oriented algorithms and hardware synthesis has been explored by Keith Hanna et al. [10], and more recently in Coq by Braibant and Chlipala [11]. Dependent types allow for very rich expressions of types, and the notion of propositions as types of the Curry-Howard correspondence means that it is then possible to express more properties than in systems which do not admit fully dependent types.

Braibant and Chlipala use the Coq language and its dependent type system as a tool to create formally verified hardware. The initial focus of verifying hardware was to verify the hardware itself, but it has recently shifted towards describing hardware in high level HDLs and verify those, and then argue that the synthesis of those descriptions are correct. Their project was to investigate verified hardware synthesis from a high-level HDL to a low-level RTL by using a certified compiler written in Coq [11]. By using a dependent type system, they can create a provably correct compiler, and have Coq generate those proofs for them for review.

Keith Hanna et al. created a specification language called Veritas+ based on type theory for specifying and reasoning about digital systems in 1990, before the popularity boom of functional programming languages [10]. In Veritas+, the notion of propositions as types and a rich dependent type system is used to describe digital systems and verifying them by inferring their correctness.

Types and type families (a kind of limited dependent types) have been used to improve Kansas Lava [12], in which they use type classes and type families to express unknown values and different ways of executing a given circuit.

Iavor Diatchki recently described a technique for integrating an SMT solver (Satisfiable Modulo Theory) [8] with the Haskell type checker (more specifically the type checker for GHC), which allows for constraints to be specified at the type level, which are then solved by the SMT solver. Utilizing this technique and using a rich type system for the expression of constraints on non-functional properties of hardware-oriented algorithm descriptions, a description can be solved by the solver to get a description that fulfills those constraints.

João Pizani et. al have implemented a library that uses dependent types for describing circuits in their work on Π-ware (Pi-ware)[13]. Their focus was on using the dependent types to verify the circuits to be correct and equivalent to simpler implementation of the same circuit, and also for automatically generating a circuit from a more high level implementation correct "up to specification".

Juurian Hage et. al have been working on improving type error messages for constraints in functional languages, and have added functionality to the GHC Haskell compiler for extending the GHC error messages with your own error messages [14], and will be of great help with informing the users of our libraries of how they violated the constraints, with more explicit (and more domain specific) error messages than the default message that GHC gives.

Goals and challenges

The goal is to create a library of Haskell modules to teach basic digital design using types with an emphasis on FPGAs. The user writes Haskell code that imports the library, which will include types and helpers for creating a hardware-oriented description of an algorithm. These types and helpers would include a possibility of specifying non-functional properties of the description and constraints thereof, such as size/area, depth, timing or power usage. The resulting descriptions could then be passed to an SMT solver via the techniques described by Iavor Datchki [8], which can work out whether the constraints are satisfiable or not, and what values satisfy those constraints if so. If the description type checks and its constraints are satisfiable, the user can pass the description and the solutions to the constraints via a helper defined in the library to the base library for hardware design in Haskell, be it Lava, Kansas Lava or CλaSH (to be determined in the project). This helper would then drop the newly added types and use the machinery defined in the base library to generate a circuit that can be passed to and run on an FPGA.

One of the challenges involved will be exploring how dependent types can be used to aid understanding of the circuits and their behavior, and whether the approach is viable for hardware-oriented descriptions of algorithms. Choosing which non-functional property is best to describe and is most relevant to the task at hand. How to express these properties and constraints in the type system in a way that is intuitive for beginners to understand but still powerful enough to tackle the task at hand will certainly be a challenge, and creating helpers to simplify this task for readers will also be a challenge. Passing the solution to these as arguments to the base library to improve the generated circuit would be another goal, i.e. to have the types guide the circuit generation, though that presents an added challenge in itself.

Digital design interface

A simple example of the interface we would be getting at is:

```haskell
{-# LANGUAGE RankNTypes, UndecidableInstances #-}

module ADD where

import CLaSH.Prelude

import GHC.Exts (Constraint)
import Data.Kind (Type)

addT :: Unsigned 8 -> Unsigned 8 -> (Unsigned 8, Unsigned 8)
addT acc a = (o, o)
  where o = acc + a


add :: Signal (Unsigned 8) -> Signal (Unsigned 8)
add = mealy addT 0

testInput :: Signal (Unsigned 8)
testInput = stimuliGenerator (1:>2:>3:>4:>Nil)

expectedOutput :: Signal (Unsigned 8) -> Signal Bool
expectedOutput = outputVerifier (1:>3:>6:>10:>Nil)

data Property a b (p :: Nat) = Annotated (Signal a -> Signal b)

type family IsNotTooBig (p :: Nat) :: Constraint
type instance IsNotTooBig p = (p <= 2) -- Area

-- verified uses the previously defined constraint, IsNotTooBig
-- and uses it in its type so that the application does not
-- type-check unless the given property satisfies the constratin.
-- it then extracts the function, which can then be passed
-- onwards to C$\lambda$sh to generate the hardware description
-- itself.
verified :: IsNotTooBig p => Property a b p ->  Signal a -> Signal b
verified (Annotated func ) = func

addP :: Property (Unsigned 8) (Unsigned 8) 2
addP = Annotated add
```

```haskell
addP2 :: Property (Unsigned 8) (Unsigned 8) 3
addP2 = Annotated add


-- This definition of a topEntity type-checks, since
-- addP has 3 as the value of its property.
topEntity :: Signal (Unsigned 8) -> Signal (Unsigned 8)
topEntity = verified addP

-- Fails with
-- ADD.hs:57:14: error:
--     • Couldn't match type ‘'False’ with ‘'True’
--         arising from a use of ‘verified’
--     • In the expression: verified addP2
--       In an equation for ‘topEntity2’: topEntity2 = verified addP2
-- Failed, modules loaded: none.

-- topEntity2 :: Signal (Unsigned 8) -> Signal (Unsigned 8)
-- topEntity2 = verified addP2
```

As we can see, the error `Couldn't match type ‘'False’ with ‘'True’` has about little information as there is, and this would need improving. The idea would be to provide a descriptive type for the desired circuit, and having the error message relay tips and pointers to the student using the library.

Type-directed search to fill holes

Another desirable property would be using typed holes to help the student and guide him along.

```haskell
combed :: Property (Unsigned 8) (Unsigned 8) 3
combed = _a
```

Currently, this gives us an error of

```
ADD.hs:45:10: error:
    • Found hole: _a :: Property (Unsigned 8) (Unsigned 8) 3
      Or perhaps ‘_a’ is mis-spelled, or not in scope
    • In the expression: _a
      In an equation for ‘combed’: combed = _a
    • Relevant bindings include
        combed :: Property (Unsigned 8) (Unsigned 8) 3
          (bound at ADD.hs:45:1)
```

If we would be able to extend that with more information on the relevant bindings, such as

```
Relevant bindings include
    Add.addP2 :: Property (Unsigned 8) (Unsigned 8) 3
```

this would help guide the student towards the right solution, by reminding him of what functions are available for use.

This mechanism would also benefit Haskell in general. A similar system exists for PureScript, a Haskell like language that compiles down to JavaScript. Adapting their solution would probably make for a good first attempt at a similar mechanism. In all likelihood it will only work for the exact type in question at first, but if we could extend it to take constraints into account, it would prove immensely useful for our library. This would also enable us to import a library of functions that have some type and property, and allow us to filter immediately out those functions that don't fit our constraints, making libraries of hardware oriented algorithms easier to use. Implementing it would probably entail extending GHC.

Work on this feature has already been started by me, and a patch with the basic functionality has already been accepted and is ready to land in GHC. The current patch only show information on bindings that fit the given hole exactly, i.e. have the same type that the hole has. We'd like to extend this functionality to also show things in scope that fit the constraints the hole imposes, so that more functions than just exact matches are displayed.

The current functionality is shown below (the "Valid substitutions include ..." message was added by me to GHC).

As an example, consider

```
ps :: String -> IO ()
ps = putStrLn

ps2 :: a -> IO ()
ps2 _ = putStrLn "hello, world"

main :: IO ()
main = _ "hello, world"
```

The results would be something like

```
• Found hole: _ :: [Char] -> IO ()
• In the expression: _
  In a stmt of a 'do' block: _ "hello, world"
  In the expression:
    do _ "hello, world"
• Relevant bindings include
    main :: IO () (bound at test.hs:13:1)
    ps :: String -> IO () (bound at test.hs:7:1)
    ps2 :: forall a. a  -> IO () (bound at test.hs:10:1)
  Valid substitutions include
    putStrLn :: String
```

```
            -> IO () (imported from ‘Prelude’ at test.hs:1:1-14
                      (and originally defined in ‘System.IO’))
putStr :: String
          -> IO () (imported from ‘Prelude’ at test.hs:1:1-14
                    (and originally defined in ‘System.IO’))
```

Type-checker plugins

Since Haskell does not have full dependent typing, a concern that arises is that we might not be able to constrain our functions well enough that we gain anything by taking the typed approach. However, GHC recently added type-checker plugins, that one can use to solve constraints in types via a plugin interface that plugs in to the type checker in Haskell.

A type-checker plugin to solve hardware related constraints would make for a good addition to the library, and allow more complex constraints to be used than what Haskell currently allows.

Tools

Another problem that the student is likely to encounter is a problem of tooling. The current tooling for taking a hardware description such as VHDL and transforming it into something that can be run directly on an FPGA is a long process, involving various GUIs with little automation possible. Adding a tool to the library for quickly going from the Haskell description of the algorithm to running on the FPGA would help the student connect his work with actual results, and allow for easier testing.

Since the library is primarily aimed at students, we could also work with very simple FPGAs such as the iCE40, which has a fully open-source toolchain (IceStorm) and development board available. [15].

Approach

The library will be written in Haskell, a standardized, general-purpose purely functional programming language, with non-strict semantics and strong static typing. One of the advantages of using a host language as powerful as Haskell is that we can make use of it to generate circuits as seen in the work of Mary Sheeran on clever circuits [16]. We would also be leaning on prior work on verification and synthesis of hardware-oriented descriptions of algorithms in Haskell in libraries such as Lava, Kansas Lava and Cλash [4][5][6].

The library will initially be based on Cλash, which takes a very high level approach to hardware oriented descriptions of algorithms. In Cλash, the hardware oriented descriptions are very similar to how one would write the code in Haskell itself. This allows us to easily tap into the type checking machinery of Haskell, without having to worry too much about how it compiles down to a hardware description.

Due to the danger of the dependent typing getting in the way of understanding the concepts instead of helping, we will only focus on a single non-functional property initially. This could for example be the size/area of the circuit, its depth or its power consumption. A study on which property should be encoded in the types would be the first part of the project. A restriction to a single non-functional property ensures that the dependently type part remains practical, manageable and helpful for a beginner.

We will restrict the library to data paths only, i.e. combinational algorithms implemented directly in hardware such as adders and multipliers. This reduces the scope from trying to tackle the entirety of hardware design (including control oriented circuits) to the more reasonable scope of algorithms and algorithm design. The library will remain useful, as there is a wide class of algorithms can be described directly in hardware as data paths.

Evaluation

The library will be evaluated based on its ability to help in the design and analysis in sufficiently interesting case studies on common hardware-oriented algorithm descriptions.

One such study would be on building adders and multipliers, a common component of many circuit designs. We would then start with implementing adders, then on to multipliers.

After implementing adders and multipliers, we coud extend the study by implementing a modular adder and multiplier (for computing $a \cdot b \mod n$ for some $a, b \in \mathbb{Z}, n \in \mathbb{Z}_+$).

References

[1] Wikipedia, "Field-programmable gate array — wikipedia, the free encyclopedia," 2016. [Online; accessed 1-December-2016].

[2] Amazon, "Amazon EC2 F1 Instances," 2016. [Online; accessed 10-December-2016].

[3] Microsoft, "Project Catapult," 2016. [Online; accessed 10-December-2016].

[4] A. Gill, T. Bull, G. Kimmell, E. Perrins, E. Komp, and B. Werling, "Introducing Kansas Lava," in *IFL*, pp. 18–35, Springer, 2009.

[5] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: Hardware design in Haskell," in *ACM SIGPLAN Notices*, vol. 34, pp. 174–184, ACM, 1998.

[6] C. Baaij, M. Kooijman, J. Kuper, A. Boeijink, and M. Gerards, "Cλash: structural descriptions of synchronous hardware using haskell," in *Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on*, pp. 714–721, IEEE, 2010.

[7] R. Eisenberg, "Dependent types in Haskell: Theory and practice." PhD thesis draft, University of Pennsylvania, 2016.

[8] I. S. Diatchki, "Improving Haskell types with SMT," in *ACM SIGPLAN Notices*, vol. 50, pp. 1–10, ACM, 2015.

[9] N. A. Danielsson, "Lightweight semiformal time complexity analysis for purely functional data structures," in *ACM SIGPLAN Notices*, vol. 43, pp. 133–144, ACM, 2008.

[10] F. K. Hanna, N. Daeche, and M. Longley, “Veritas+: a specification language based on type theory,” in *Hardware specification, verification and synthesis: mathematical aspects*, pp. 358–379, Springer, 1990.

[11] T. Braibant and A. Chlipala, “Formal verification of hardware synthesis,” in *International Conference on Computer Aided Verification*, pp. 213–228, Springer, 2013.

[12] A. Gill, T. Bull, A. Farmer, G. Kimmell, and E. Komp, “Types and associated type families for hardware simulation and synthesis,” *Higher-Order and Symbolic Computation*, pp. 1–20, 2013.

[13] J. Pizani, W. Swierstra, and Y. Sijsling, “Π-ware: Hardware description and verification in agda,” in *21st International Conference on Types for Proofs and Programs*, pp. 1–26, Leibniz International Proceedings in Informatics, 2015.

[14] J. Hage, “Domain specific type error diagnosis (domsted),” 2014.

[15] C. Wolf and M. Lasser, “Project icestorm.” `http://www.clifford.at/icestorm/`. [Online; accessed 23-March-2017].

[16] M. Sheeran, “Generating fast multipliers using clever circuits,” in *International Conference on Formal Methods in Computer-Aided Design*, pp. 6–20, Springer, 2004.