On Applications of Dependent Types to Parameterised Digital Signal Processing Circuits

Craig Ramsay
University of Strathclyde
Glasgow, Scotland
craig.ramsay.100@strath.ac.uk

Louise H. Crockett
University of Strathclyde
Glasgow, Scotland
louise.crockett@strath.ac.uk

Robert W. Stewart
University of Strathclyde
Glasgow, Scotland
r.stewart@strath.ac.uk

***Abstract*—We explore the use of dependent types to address the disparity between the theory and the practical hardware description of DSP circuits. After discussing an approach to modeling synchronous circuit behaviour in Idris (a pure functional language with dependent types), two DSP case studies are introduced — an FIR filter with optimal wordlengths and a CIC decimator with register pruning. Both of these scenarios prove difficult to describe in a parameterised fashion using traditional HDLs and, as such, many implementations rely on ad hoc circuit generators which are challenging to test and evaluate. This work demonstrates that such circuits are readily described in an environment with dependent types. Dependent types can also encode various contracts between the IP designer and its user. These contracts are automatically verified by the Idris type checker before compilation, precluding many common mistakes in IP development and evaluation.**

I. Introduction

When describing Digital Signal Processing (DSP) circuits, there is often a disparity between the generalised mathematical theory and the practical description in a Hardware Description Language (HDL). HDLs such as VHDL and Verilog do not provide the language features required to fully parameterise some simple structures, such as direct-form Finite Impulse Response (FIR) filters with optimal wordlength growth.

As a consequence, designers either hand-craft an ad hoc model for their own parameters or turn to software programming, in an attempt to generate hardware descriptions in a fully parameterised way. Perl has been a common choice for such a task, however this offers extremely little in terms of static checking of the circuit generators — placing an extra burden of testing on the designer. This can also be problematic when evaluating 3rd party Intellectual Property (IP); evidence of testing is vital in the absence of any invariants checked by a compiler. The user of these IPs will need to know what constitutes a valid set of parameters, which edge cases have been handled, and any assumptions that have been made. This issue can even present itself in many functional HDLs with expressive type systems, such as [1]–[3].

This paper introduces pure, functional programming languages *with dependent types* as a solution to describing parameterised DSP circuits faithfully, and with structures verified at compile-time. Automatically verifying these properties provides a contract between the IP developer and the user; the developer must generate well-formed circuits for all possible parameter sets, and the user is given a clear, computer-checked proof of this behaviour. Dependent types allow the *type* of a program (in this case, encoding the structure of a circuit) to depend on the *values* of its arguments, not just the argument types (e.g. a coefficient's value, not just its wordlength). We use the language Idris [4] to make this demonstration of modeling DSP circuits. Currently this is in simulation only, but synthesis remains a promising avenue for future work given the success of similar synthesisable languages without dependent types [1]–[3] and dependently typed tools from other domains [5].

II. Modelling synchronous circuits in Idris

To model simple circuit behaviour in any language, it is necessary to have fixed-size data types and a means of describing synchronous signals. These topics have considerable complexity when considering circuit synthesis — but much of this can be neglected when performing simulation alone. One model of synchronous signals is an infinite stream where the k^{th} element represents the discrete-time sample that is stable during the k^{th} clock cycle. This technique can be seen in various forms in languages such as Kansas Lavas [1] and CλaSH [2], both hosted in Haskell.

The host language must have support for *lazy evaluation* if using these infinite streams since we, unfortunately, run our simulation on computers with finite memory. One subtlety arising from this approach is that it is possible to describe a variety of non-synthesisable circuits. For example:

- Dropping an element from the stream describes a time advance, and is non-causal.

```
adv : Stream a -> Stream a
adv (x :: xs) = xs
```

- Some recursive uses of streams would infer circuits with infinite memory resources [2].

```
elephant : a -> Stream a -> Stream a
elephant i (x :: xs) = i :: x :: elephant i xs
```

Such descriptions can be precluded by hiding the Stream implementation and only exposing "safe" functions to operate on these streams — such as `delay`, and a functor or applicative interface.

Sections III and IV continue by considering how Idris' dependent types can be used to describe and reason about commonplace DSP circuits more precisely than traditional HDLs.

III. First steps: Minimal Bit Growth for FIR Adder Chains

Consider the direct form FIR filter shown in Figure 1. All wordlengths have been annotated with the worst-case for each unsigned arithmetic operation in isolation. Note that we have adopted unsigned arithmetic for the purposes of illustrating Idris concepts, acknowledging that signed arithmetic would be preferred for FIR filter implementation. The interested reader can access our full source, including a signed variant, at [6].

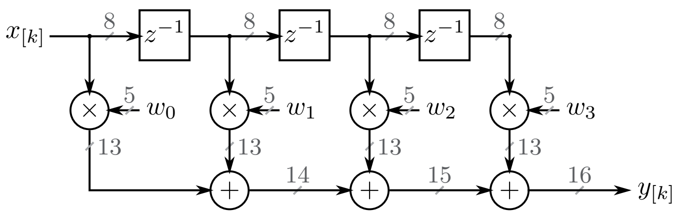

Fig. 1. A direct form FIR filter with worst-case growth along the adder chain

For this introductory example, the input is 8_b (an 8-bit word), the coefficients are all 5_b, and the adder chain ends as 16_b. Given n_b and m_b unsigned words, multiplication with worst-case bit growth is represented by $(n+m)_b$, and addition gives $(\text{Max}[n, m] + 1)_b$. This sort of growth can be captured by VHDL designs using generics and `for generate` statements. (Note that there is no type inference, however, so every intermediate signal must be explicitly defined.). As an introduction to Idris' syntax we present this worst-case bit growth for arithmetic functions in Listing 1, where the type `Unsigned n` represents an unsigned integer of n bits.

Listing 1. Worst-case bit growth for binary arithmetic functions

```
mul : Unsigned n -> Unsigned m -> Unsigned (n+m)
mul (U a) (U b) = U (a * b)

add : Unsigned n -> Unsigned m
   -> Unsigned (max n m + 1)
add (U a) (U b) = U (a + b)
```

Each function has a type (after "`:`"), and an implementation (after "="). For clarity, the type of the multiplication function, `mul`, should be read as "a function accepting arguments of type `Unsigned n` and `Unsigned m`, and returning a value of type `Unsigned (n+m)`".

In the case of Figure 1, a circuit can be described with wordlengths better than the worst-case for two reasons:

1) Each operation was considered in isolation but repeated additions will accumulate quantisation effects when the range of a number does not align with powers of 2. For example, y in Figure 1 will only inhabit values within the range $[\![0, 2^{15} - 1]\!]$, despite its 16_b annotation.
2) The coefficients will often be constants. In this case, the bit growth due to multiplication will vary with the numerical value of each constant coefficient.

The latter is particularly relevant, as it clearly demands a language with dependent types — a *term*-level value (a coefficient) must be used to compute a *type* (the output wordlength). The next section applies Idris to these challenges.

A. An Idris Implementation

Better bit growth is facilitated by types that track the integer range each signal can inhabit, rather than immediately rounding to the required number of bits (i.e. $\lceil log_2(range)\rceil$). Our `Bounded` type implements this, where a number of type `Bounded n` is in the closed interval $[\![0, n]\!]$ (i.e. any value between 0 and n, inclusive). Listing 2 shows two arithmetic functions on `Bounded` that help ensure minimum bit growth for the FIR filter example — `mulConst` to multiply a `Bounded` with a constant, and `add` to add two `Bounded`s.

Listing 2. Minimum bit growth for `Bounded` arithmetic functions

```
mulConst : Bounded n -> (m: Nat) -> Bounded (n*m)
mulConst (B x) m = B (x*m)

add : Bounded n ->Bounded m -> Bounded (n+m)
add (B x) (B y) = B (x+y)
```

Note `mulConst`'s use of dependent types, where the *type* of the output depends on the *value* of an argument. A full FIR filter circuit can be constructed using these arithmetic functions, as it is just a dot product of j coefficients (w) and the last j samples of a discrete time signal (x).

$$y_{[k]} = \sum_{i=0}^{j-1} w_i \cdot x_{[k-i]} \tag{1}$$

To construct the type for the dot product's output, consider the worst-case magnitude of each term in Eq. 1. As the range of $x_{[k]}$ is constant for all k, the type can become:

$$|y_{[k]}| = |x_{[k]}| \sum_{i=0}^{j-1} |w_i| \tag{2}$$

From this, we can deduce that a valid type for the dot product function is `Bounded (n * sum ws)`, given a collection of j coefficients, `Vect j Nat` called `ws`, and a collection of j samples of x, `Vect j (Bounded n)`. Note that `sum` is an ordinary function and it is a consequence of dependent types that we can use it to construct a type for the dot product output. Listing 3 shows the implementation of this combinatorial dot product and the FIR model that "lifts" this dot product into a `Stream`, modelling synchronous logic.

Notice that line 7 includes a "rewrite" rule that seems extraneous to the model of the circuit. This acts as a small proof for the Idris compiler, demonstrating that the implementation does agree with the type we described. The type of the dot product is defined as in Eq. 2, and while our recursive implementation

Listing 3. Dependently typed FIR implementation

```
dotProd : (ws : Vect j Nat)
       -> Vect j (Bounded n)
       -> Bounded (n * sum ws)
dotProd {j=Z}          _        _        = zeros
dotProd {j=S l} {n} (w :: ws) (x :: xs) =
  let y = add (mulConst x w) (dotProd ws xs)
  in  rewrite dotProdDistrib n w l ws in y

fir : (ws : Vect j Nat)
   -> Stream (Bounded n)
   -> Stream (Bounded (n * sum ws))
fir {j} ws x = liftA (dotProd ws) (window j zeros x)
```

is mathematically equivalent, it does have a subtly different structure, as shown in Eq. 3 for the s^{th} recursive step.

$$|y_{[k]}| = |x_{[k]} \cdot w_s| + |x_{[k]}| \sum_{i=s+1}^{j-1} |w_i| \tag{3}$$

The rewrite rule is reminding the compiler of multiplication's distributive property, and thus Eq. 2 $\equiv$ Eq. 3.

In summary, dependent types have been used to implement an FIR filter in Idris that models *minimal* bit growth (hence minimal resources) based on the constant values of the coefficients. As we track the ranges in the types, the compiler ensures that we implemented the minimal growth at each arithmetic stage, as defined by our contract's specification in Eq. 2. If an implementation fails to do so for *any* possible set of parameters, it will raise a compiler error for both the IP designer and the IP user. The compiler will disallow designs that, for example, do not grow wordlengths for arithmetic or do not sensibly use each coefficient exactly once, giving us some confidence that we have correctly implemented a form of dot product. Note that this example's types encode wordlengths and do not strictly guarantee the arithmetic meaning of the output or its dependence on time, but the patient designer can refine these types to do so when deemed necessary. Further discussion of this possibility is presented in Section V.

B. Comparisons to existing HDLs

Compare this dependently typed, minimum bit growth FIR filter to the implementations possible in other HDLs. A typical VHDL FIR filter can be parameterised in terms of its coefficients, the wordlength of the coefficients, and the input wordlength. However, the bit growth is likely to be worse than even the scenario presented in Figure 1. Because of the lack of type inference or type-level generate statements in VHDL, a common approach is to simply resize all arithmetic stages to match the full precision output — heavily relying on synthesis tools to remove unused nets.

Although this is a valid design choice when considering the filter in isolation, it presents practical difficulties for real designs. The filter will usually be just one part of a larger chain of DSP circuits. At several points along the data path, the full precision signals will be shortened to constrain resource usage. In this case, the designer may employ two strategies:

- Truncation/rounding of the LSBs.
- Removing uninhabited MSBs identified by Eq. 2.

The second option is appealing as it can reduce wordlengths without loss in precision, but it requires extra manual effort (for each coefficient set!) just to emulate a static property of our Idris implementation. Beyond this, breaking the reliance on synthesis tools allows the designer to reason about resource usages directly from the source — including how resource usage mathematically relates to any design parameters.

There are also clear benefits above other modern functional HDLs, such as Lava [1]. In Lava, a similar circuit can be described using dynamically sized lists of bits to represent each word. It is then the execution of a (software) Haskell program that generates the circuit, since statically sized structures are required for most structural hardware descriptions. In this case, the output circuit *might* be equivalent to the Idris implementation, but there are no guarantees about wordlengths checked by the compiler — this is what we have addressed with dependent types. Similar benefits are demonstrated in an adjacent domain; the language Proto-Quipper-D uses dependent types to ensure correctness of entire "families" of parameterised quantum circuits [5]. Without these compiler checks, there is a large burden on the developer to provide good evidence of testing.

IV. Going further: Pruning in CIC Interpolators/Decimators

Moving away from simple FIR examples now, it is natural to start exploring other DSP applications that are parameterised slightly differently. We start to notice that many of these do place unmet demands on existing HDLs in their most general form. A noteworthy example is the Cascaded Integrator-Comb (CIC) decimator/integrator, as these designs cannot rely on synthesis tools to mask imprecise descriptions from traditional HDLs.

CIC decimation filters are often used as a very low resource means of resampling — composed of a chain of N integrators, followed by a $\frac{1}{R}$ downsampler, followed by N comb filters with a differential delay of M. Figure 2 shows an example CIC decimator with $R = 8$, $N = 3$ and $M = 1$.

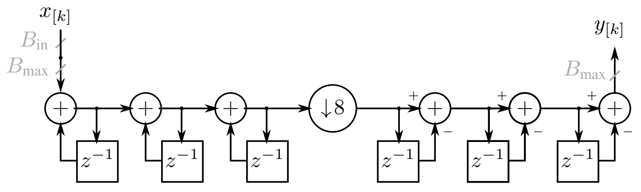

Fig. 2. A CIC decimator without pruning. ($R = 8$, $N = 3$ and $M = 1$)

Hogenauer introduced a register pruning technique for CIC filters [7], deriving equations for the mean error and variance introduced by truncation at each stage. It is suggested that, given a desired output wordlength, a legitimate design choice is to prune the wordlengths of all previous stages maximally without accumulating an error greater than that introduced by the final rounding/truncation. This choice results in Eq. 4, describing the number of LSBs to discard at the j^{th} stage.

$$B_j = \left\lfloor -\log_2 F_j + \log_2 \sigma_{T_{2N+1}} + \frac{1}{2}\log_2 \frac{6}{N} \right\rfloor \quad (4)$$

where F_j is the variance error gain for the j^{th} stage, $\sigma_{T_{2N+1}}$ is the total variance at the output due to truncation, and N is the number of stages. Note that the first two terms have complicated definitions of their own, including cases, sums, exponentials, and binomial coefficients [7].

As the pruned bits do contain information, synthesis tools cannot perform an equivalent optimisation given an non-pruned description. We apply the same techniques as shown in Section III to implement a fully parameterised, pruned CIC decimator. Our implementation (omitted for brevity, but available in full at [6]) makes use of Idris' single language that can be used at the term-level and the type-level. All language constructs can be used to implement Eq. 4, and this can then be used to direct the type of each stage in a CIC implementation. Using Eq. 4 at the type-level gives us the contract between IP designer and user, which is verified by Idris at compile time.

Although this early work is in simulation only, we can still explore post-layout results for this CIC pruning algorithm using an ad hoc Verilog/VHDL generator. Fig. 3 presents the LUT usages for different CIC filter parameters, highlighting the resource savings attained by Hogenauer pruning. While these results are generated with a CIC utility from the KiwiSDR project [8] (a C program that returns Verilog code), the outputs are structurally identical to our Idris implementation — only without the safety of our dependently typed properties.

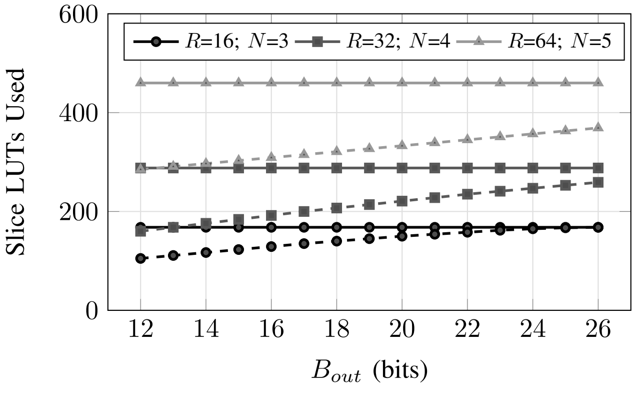

Fig. 3. Post-layout results for ad hoc CIC filter generation with Hogenauer pruning (dashed lines) and without (solid lines). $M = 1$ and $B_{in} = 16$ for all lines.

Note that the approach to circuit generation in [8] is a prime example of some common issues that we are trying to address with dependent types. There is no verified contract between the IP designer and user for the range of valid parameters, their effect on the generated circuit, and no clear evidence of testing. One such bug found while generating Fig 3 is that any B_{out} which requires bit extension at the final stage, rather than truncation, will silently generate invalid Verilog.

While both of our examples focus on computing type-level wordlengths, the techniques can be applied to many other type-level constructs — e.g. circuit topology. It is only a small leap to use these techniques to write type-safe descriptions of many other circuits. We will explore this opportunity further in Section V.

A. Comparisons to existing HDLs

Idris gives us the same advantages over VHDL here as in Section III, with the addition that VHDL's limitations cannot be partly mitigated by optimisations built into synthesis tools.

In comparison to HDLs embedded in Haskell, such as Lava, a similar structure may be technically realisable with type-level functions and singletons. However, Haskell does not offer the luxury of one rich language for both the term and type levels. Because of this, implementing the complex, type-level function required to represent Eq. 4 would prove to be a particularly challenging excursion.

V. Further Applications

We have introduced two example use cases for Idris as an environment for simulating (and, eventually, synthesising) DSP circuits. These are both introductory examples, focusing only on type-level wordlengths in commonplace DSP structures. Encoding wordlengths in a circuit's type was chosen here because it is both:

1) A genuinely valuable property to verify for DSP circuits. When left unchecked, wordlengths can be a common source of "off by one" errors and present ambiguity in arithmetic modes (extending, saturating, or wrapping). They can also hide valuable design information from the user, as we have seen in our FIR example.
2) An approachable first introduction to type-level computations, the classes of error that Idris' type system can identify statically, and allows us to very briefly touch on the use of dependent types as a general proof assistant.

However, as hinted at previously, dependent types actually offer a user the means to track many other circuit generator properties in their types too — not just wordlengths. Remember that if we encode such a property in our circuit's type, the compiler will mechanically check that this holds for the entire parameter space. As one example, [9] introduces an alternative encoding of binary numbers in Idris which carries the natural number representation of a binary word in its type. Here we can define the arithmetic intent of a circuit using natural numbers and have the compiler ensure that our binary implementation preserves this (i.e. it is *functionally correct*).

Our proposal allows the developer to program (without ad-hoc compiler support) using as much type-level safety as is desired — anywhere from checking wordlengths, to ensuring pipeline depths of two branches are equal, or even to fully verifying the arithmetic meaning of a family of circuits.

Speculating about futher applications for DSP circuits, consider the literature's rich set of digital design techniques that take simple (inefficient) operations and restructure them into cheaper or more amenable forms. Two such examples are area-efficient, parallel FIR structures for high-throughput applications (such as the Fast FIR Algorithm [10]) and Multiple Constant Multiplier blocks for multiplierless arithmetic in filters (such as Hcub [11]). Both of these typically rely on software programming for generic implementation and their supporting publications include proofs that the operation is equivalent to their unoptimised counterparts. A particularly thorough implementation with dependent types could include these proofs/contracts alongside the implementation. This would ensure that the ideas are not only peer-reviewed, but have been mechanically verified. Importantly, this verification would also apply directly to their concrete implementation, not just the conceptual algorithm.

VI. Related work

We acknowledge the rich history of functional HDLs, including [1]–[3] but we focus on the narrower field of dependently typed languages. The work in [5] introduces an application of dependent types for generating families of quantum circuits, hinting towards an exciting future for synthesis of digital circuits using similar techniques. Ref. [12] introduces Π-ware, an extremely low-level structural HDL with dependent types (embedded in Agda), but does not address its application in a wider DSP context. The authors also identify the productivity associated with Π-ware's low-level descriptions as point for improvement. We instead advocate for future work exploring a new dependently typed language with the higher-level functional productivity of CλaSH [2].

VII. Conclusions

This work explored how dependent types can be used to represent fixed-point structures common to many DSP circuits. The best implementations of these structures to date have used ad hoc software that generates a circuit during its run-time (rather than a direct hardware description). We have shown that dependent types allow a compiler to check these structures, rather than suppressing bugs until run-time with a particular set of parameters. This method of hardware description minimises the burden of testing for designers of reusable IP, as many of the structural properties of a circuit can be guaranteed by their type, and makes it possible for users to confidently evaluate IPs. These implementations often require no extra effort, and better still, using such specific types can even help guide the implementation of complex structures.

Acknowledgement

The authors would like to thank Xilinx for supplying EDA tools for this project and acknowledge funding for Craig Ramsay under EPSRC grant no. EP/N509760/1. All source code underpinning this publication is openly available at [6].

References

[1] A. Gill, T. Bull, G. Kimmell, E. Perrins, E. Komp, and B. Werling, “Introducing Kansas Lava,” in *Proceedings of the Symposium on Implementation and Application of Functional Languages*, ser. LNCS, vol. 6041. Springer-Verlag, Sep 2009.

[2] C. Baaij, “Digital circuit in cλash: functional specifications and type-directed synthesis,” Ph.D. dissertation, University of Twente, Netherlands, 1 2015, eemcs-eprint-23939.

[3] M. Sheeran, “ufp a algebraic vlsi design language,” OUCL, Tech. Rep. PRG39, November 1983.

[4] E. Brady, “Idris - systems programming meets full dependent types,” in *PLPV'11 - Proceedings of the 5th ACM Workshop on Programming Languages Meets Program Verification*, 01 2011, pp. 43–54.

[5] P. Fu, K. Kishida, and P. Selinger, “Linear dependent type theory for quantum programming languages,” in *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*, ser. LICS '20, 2020, p. 440–453. [Online]. Available: https://doi.org/10.1145/3373718.3394765

[6] C. Ramsay. (2020) Source code for: "on applications of dependent types to parameterised digital signal processing circuits". [Online]. Available: https://doi.org/10.15129/db040cb9-9e48-4823-8616-cbc0ace1b6cd

[7] E. Hogenauer, “An economical class of digital filters for decimation and interpolation,” *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 29, no. 2, pp. 155–162, 1981.

[8] J. Seamons. (2014) Cic filter register pruning utility. [Online]. Available: http://www.jks.com/cic/cic.html

[9] E. Brady, J. McKinna, and K. Hammond, “Constructing correct circuits: Verification of functional aspects of hardware specifications with dependent types,” in *Proceedings of the Eighth Symposium on Trends in Functional Programming, TFP 2007, New York City, New York, USA, April 2-4. 2007.*, 2007, pp. 159–176.

[10] D. Parker and K. Parhi, “Area-efficient parallel fir digital filter implementations,” in *Proceedings of International Conference on Application Specific Systems, Architectures and Processors: ASAP '96*, 1996, pp. 93–111.

[11] Y. Voronenko and M. Püschel, “Multiplierless multiple constant multiplication,” *ACM Trans. Algorithms*, vol. 3, no. 2, p. 11–es, May 2007. [Online]. Available: https://doi.org/10.1145/1240233.1240234

[12] J. P. P. Flor, W. Swierstra, and Y. Sijsling, “Pi-Ware: Hardware Description and Verification in Agda,” in *21st International Conference on Types for Proofs and Programs (TYPES 2015)*, vol. 69, 2018, pp. 9:1–9:27. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2018/8479