Correctness Properties for Iterated Hardware Structures

Phillip J. Windley

Laboratory for Applied Logic, 3370 TMCB, Brigham Young University, Provo UT 84602
Phone: 801.378.3722, FAX: 801.378.7775, email: windley@cs.byu.edu

Abstract Iterated structures occur frequently in hardware. This paper describes properties required of mathematical relations that can be implemented iteratively and demonstrates the use of these properties on a generalized class of adders. This work provides a theoretical basis for the correct synthesis of iterated arithmetic structures.

1 Introduction

Iterated structures occur frequently in hardware. Often these structures are used to implement arithmetic operations. While the intuition behind why such operations can be implemented through iteration is well understood, correctly synthesizing iterated structures requires a well–developed mathematical theory of why such structures are correct. To be generally useful, this theory should be as general as possible.

In an effort to demonstrate dependent types, Keith Hanna developed a theory of iterated hardware structures in[4]. While Hanna's work sets out the direction such generalized theories should take, his theories apply only to iterated cells with only one input and one output.

This paper presents an extension to Hanna's work by generalizing the theory for i inputs and j outputs. In addition, we have completed our work in the HOL theorem prover [3] which is based on simple type theory. The use of simple type theory for this work, rather than the more complicated, and undecidable, dependent type theory of [4] means that users of our theory incur fewer proof obligations regarding types. We demonstrate the use of the theory by developing a general theory of adders.

2 A Brief Introduction to HOL.

To ensure the accuracy of our specifications and proofs, we developed them using a mechanical verification system. The mechanical system performs syntax and type checking of the specifications and prevents the proofs from containing logical mistakes. The HOL system was selected for this project because is has higher-order logic, generic specifications and polymorphic type constructs. These features directly affect the expressibility of the specification language. Furthermore HOL is widely available, robust, and has a growing world-wide user base. However, there is nothing our work that requires the HOL theorem proving system.

HOL is a general theorem proving system developed at the University of Cambridge [1, 2] that is based on Church's theory of simple types, or higher-order logic. Although Church developed higher-order logic as a foundation for mathematics, it can be used for reasoning about computational systems of all kinds. Similar to predicate logic in allowing quantification over variables, higher-order logic also allows quantification over predicates and functions thus permitting more general systems to be described.

HOL is not a fully automated theorem prover but is more than simply a proof checker, serving as a proof assistant. HOL has several features that contribute to its use as a verification environment:

- Several built–in theories, including booleans, individuals, numbers, products, sums, lists, and trees. These theories build on the five axioms that form the basis of higher-order logic to derive a large number of theorems that follow from them.
- Rules of inference for higher-order logic. These rules contain not only the eight basic rules of inference from higher–order logic, but also a large body of *derived* inference rules that allow proofs to proceed using larger steps. The HOL system has rules that implement the standard introduction and elimination rules for Predicate Calculus as well as specialized rules for rewriting terms.
- A large collection of tactics to support goal directed proof. Included in HOL are tactics which rewrite a goal according to some previously proven theorem or definition, remove unnecessary universally quantified variables from the front of a goal, and split equalities into two implicative subgoals.
- A proof management system that keeps track of the state of an interactive proof session.
- A metalanguage, ML, for programming and extending the theorem prover. Using the metalanguage, tactics can be put together to form more powerful tactics, new tactics can be written, and theorems can be aggregated to form new theories for later use. The metalanguage makes the verification system extremely flexible.

For the most part, the notation of HOL is that of standard logic: $\forall$, $\exists$, $\wedge$, $\vee$, etc. have their usual meanings. There are a few constructs that deserve special attention due to their use in the remainder of the paper:

- HOL types are identified by a prefixed colon. Built–in types include `:bool` and `:num`. Function types are constructed using $\longrightarrow$. HOL is polymorphic; type variables are indicated by a type names beginning with an asterisk.
- The HOL conditional statement, written `a` $\rightarrow$ `b` | `c`, means "if `a`, then `b`, else `c`."
- Tuples are formed using a comma. Parentheses are only required when the scope of the comma is ambiguous.
- The HOL list containing elements `a`, `b`, `c`, and `d` is represented as `[a;b;c;d]`. A list that contains elements with type `x` has the type `:(x)list` where `x` can be any valid type (including type variables since HOL is polymorphic).
- The function `MAP f lst` applies the function `f` to each member of the list `lst` returning a new list made from the results of each application.

Table 1: Restricted types used in iterated hardware structures.

Name	Meaning
`pos`	positive natural numbers
`nat`	natural numbers
`N n`	natural numbers less than n
`bit`	bits (i.e. 0 or 1)
`bvec`	bit vectors
`V m`	vectors with elements less than m
`vec (P,l)`	vectors of length l with elements all meeting the predicate P

3 Numbers and Numerals

Iterated arithmetic hardware structures are based on the relationship between numbers and numerals. We expect that the behavior of a hardware device will be to perform an arithmetic operation on numbers and produce numbers as the result. We implement such structures by iterating devices which can be thought of as working on the separate digits[1] of a numeral representing the number. This is true whether we are iterating a 1–bit adder 32 times to produce a device that adds two numbers (with magnitude less than 2^{32}) or iterating an 8–bit carry lookahead adder 4 times to get the same result. In the latter case, one must consider the numeral to have four digits of magnitude less than 256

4 Type Restrictions

In the definitions and theorems that follow we will frequently use type restrictions to restrict the the value that a variable may take. These restrictions are expressed as predicates. Inside quantification, we will append these predicates to the variable that they modify with two colons. So, ∀ `v::P. B` is syntactic sugar for ∀ `v. P` ⟹`B`. One can, for most purposes, however, read the term ∀ `v::P. B` as "for all v of type P..." We will use the restricted types defined in Table 1 throughout the rest of this paper.

5 A Generalized Theory of Iteration

Figure 1 shows how iterated hardware structures can be defined recursively. In this case an $n+1$–bit structure is implemented by a 1–bit cell and an n–bit cell. Textually, we can represent this same process (in a more general way) as follows:

```
(fold 0 cell in out cin cout = (cin = cout)) ∧
(fold (SUC n) cell in out cin cout =
    ∃ e . (fold n cell in out cin e) ∧
          (cell (MAP (λf.f n) in) (MAP (λf.f n) out) e cout))
```

[1]In this paper, we will refer to the seperate components of a numeral in any base as digits, even though technically, they are only called this for numerals in base 10.

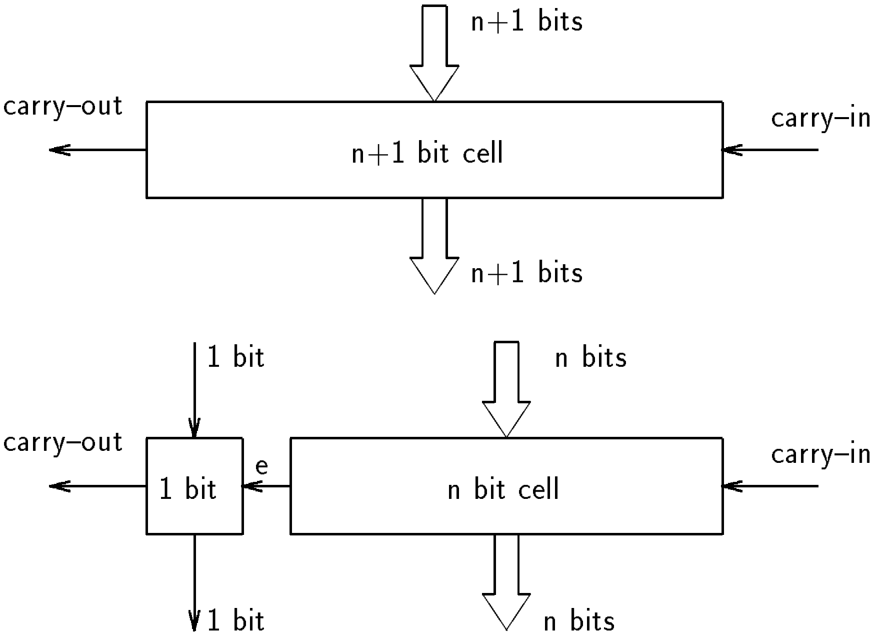

Figure 1: `fold` recursively implements iterated hardware structures.

Folding a cell 0 times results in a structure where the carry–in is wired directly to the carry–out. Folding a cell $n+1$ times is done by wiring a cell to a structure obtained by folding the cell n times. The existentially quantified variable, `e`, represents a hidden line connecting the carry–in of the most significant cell to the carry–out of the rest of the structure.

In order to make this definition as general as possible, `in` and `out` are lists of input and output busses respectively. We select the n^{th} line in the bus by applying the bus to n. Thus, `(bus n)` represents the n^{th} line in `bus`. So, the term `(MAP (λ f. f n) in)` selects the n^{th} member of each input in the list to be used by `cell`.

Using our definition of `fold`, we can define two properties that an arithmetic function (expressed in a relational form) must have in order to be iteratively implementable.

First, any arithmetic relation that we wish to implement iteratively must be *proper*.

```
⊢def proper R = ∀ a b (c:*) .
      (EVERY IS_ZERO a) ∧
      (EVERY IS_ZERO b) ⇒
      R 1 a b c c
```

The relation `R` takes a number giving the base it is to work in, 1 in this case; a list of inputs, `a`; a list of outputs, `b`; a carry–in, `c`; and a carry–out, also `c` in this instance. A relation is proper when it is true in base 1 with all its inputs and outputs are 0 and its carry–in is wired to its carry–out. This relationship, obviously, grows out of the base case in the recursive definition of `fold`. The proof that an arithmetic relation is proper is almost always trivial (see Section 5 for an example).

The second property that any arithmetic relation that we wish to implement iteratively must have is called factorizability. The following defines what it means for an arithmetic relation to be factorizable:

```
⊢def factorizable R =
     ∀ m k ::pos .
     ∀ l1 l2 ::pos .
     ∀ a0 :: vec(N m,l1) .
     ∀ b0 :: vec(N m,l2) .
     ∀ a1 :: vec(N k,l1) .
     ∀ b1 :: vec(N k,l2) .
     ∀ c d :* .
     let a = MAP (λ (x, y) . x + (y * m)) (a0, a1) and
         b = MAP (λ (x, y) . x + (y * m)) (b0, b1) in (
     (∃ e . (R k a1 b1 e d) ∧ (R m a0 b0 c e)) ⇒
     (R (k * m) a b c d))
```

The predicate is true of relations that can be implemented piecewise. That is, when a relation R on numbers less than $k \times m$ can be implemented by two instances of the same relation on numbers less than k and m respectively with a carry signal.

Using the factorization predicate, we can prove that a relation, R, that is both proper and factorizable can be iteratively implemented:

```
⊢ ∀ w .
  ∀ l1 l2 :: pos .
  ∀ m :: pos .
  ∀ (R:num -> (num)list -> (num)list -> * -> * -> bool) .
  ∀ A :: vec(V m,l1) .
  ∀ B :: vec(V m,l2) .
  ∀ c d .
  (factorizable R) ∧
  (proper R) ⇒
  ((fold w (R m) A B c d) ⇒
   (R (m EXP w) (MAP (VAL (m,w)) A) (MAP (VAL (m,w)) B) c d))
```

For a given relation R, this theorem states that when R is proper and factorizable, folding a base–m version of the relation w times implies the base–m^w version of the relation.

6 A Theory of Adders

In this section, we define an relationship for base–m addition, show that the relation is both proper and factorizable, and, using these results along with our theory, prove that an iterated adder has the correct behavior.

We can define a relation add as follows:

```
⊢def add m (w1, w2) w3 ci co =
        EVERY (N m) [w1;w2;w3] ⇒
        (w1 +  w2 + (bv ci) = w3 + (m * (bv co)))
```

In this definition, `m` is the base, `w1`, `w2`, and `w3` are numbers less than `m`, and `ci` and `co` are boolean values representing the carry–in and carry–out. The function `bv` converts a boolean value to a number. Note that the relation is defined on base–m numbers, not numerals.

We have produced specialization of the general definitions and theorems given the last section for the most frequent cases. Using a specialization for 2 inputs and 1 output, we can show that `add` is `proper`. The proof comes down (after rewriting with the definitions) to showing that the following is true:

```
(0 + (0 + (bv c)) = 0 + (1 * (bv c)))
```

This proof is, of course, trivial, and establishes that

```
⊢ proper add
```

The second obligation, factorizability, while somewhat more complicated, is still surprisingly easy to prove. After expanding the definitions and some minor manipulations, we must show that

```
"(a0 + (a1*m)) + ((b0 + (b1*m)) + (bv c)) = (out0 + (out1*m)) + ((k*m)*(bv d))"
```

under the following two assumptions:

```
    [ "a0 + (b0 + (bv c)) = out0 + (m * (bv e))" ]
    [ "a1 + (b1 + (bv e)) = out1 + (k * (bv d))" ]
```

This is, of course, true by commutivity and associativity establishing that

```
⊢ factorizable add
```

The `add` relation is used to specify the correct behavior of an adder:

```
⊢def Adder (m,l) (A, B) OUT c d =
      add (m EXP l) (VAL(m,l) A, VAL(m,l) B) (VAL(m,l)OUT) c d
```

The definition operates on base–m numerals of length l. The function `VAL` is an abstraction function that converts numerals to numbers. Since we know that

```
⊢ (VAL (m,w) a) < (m EXP w)
```

The definition of `Adder` can be expanded to

```
⊢ Adder(m,l)(A,B)OUT c d =
    ((VAL(m,l)A) + ((VAL(m,l)B) + (bv c)) =
     (VAL(m,l)OUT) + ((m EXP l) * (bv d)))
```

The implementation of an adder is done iteratively:

```
⊢def Adder_impl (m,l) (A,B) OUT c d = (fold l (add m) (A,B) OUT c d)
```

The implementation iterates a base–m adder l times. Again, after expansion, this definition is like iterative specifications of adders done without our theory:

```
⊢ ∀m A B c d OUT.
   (Adder_impl(m,0)(A,B)OUT c d = (c = d)) ∧
   (Adder_impl(m,SUC l)(A,B)OUT c d =
       (∃e. fold_21 l(add m)(A,B)OUT c e ∧
            add m(A l,B l)(OUT l)e d))
```

Using our results about the factorizability of `add` and the theorems from the last section, we can show that the iterative structure correctly implements the desired behavior:

```
⊢ Adder_impl (m,l) (A,B) OUT c d ⇒  Adder (m,l) (A,B) OUT c d
```

The proof of this theorem is trivial and consists of expanding the definitions and applying a tactic based on the results of the last section.

This theorem is general enough to describe a large class of adders. This theorem applies equally whether we are implementing a base–2^{32} adder by iterating a 2–bit adder 32 times or iterating an 8–bit carry lookahead adder 4 times. This kind of generality is important in the synthesis of correct hardware structures since it gives the designer room to maneuver in the design space, trading off space and speed, while maintaining a correct design.

We have shown how this theory can be applied to adders, but it applies equally well to any arithmetic operation that is factorizable, including multiplication, comparison, etc..

7 Conclusion

This paper has presented a generalized theory of iterated hardware structures for implementing arithmetic operations. We have described properties that determine when an arithmetic operation *can* be correctly implemented by iteration. We have demonstrated the use of the theory by developing a theory of iterated adders that is general enough to be used to describe most adders in common usage. The theory forms the basis for correctly synthesizing adders based on constraints given by the designer.

References

[1] A. Camilleri, M. Gordon, and T. Melham. Hardware verification using higher order logic. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs*. Elsevier Scientific Publishers, 1987.

[2] M. J. Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwhistle and P. Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis*. Kluwer Academic Press, 1988.

[3] M. J. C. Gordon and T. F. Melham. *Introduction to HOL*. Cambridge University Press, 1993.

[4] F. K. Hanna, N. Daeche, and M. Longley. Specification and verification using dependent types. *IEEE Transactions on Software Engineering*, SE-16(9), September 1990.