Chapter 1

Constructing Correct Circuits: Verification of Functional Aspects of Hardware Specifications with Dependent Types

Edwin Brady[1], James McKinna[1], Kevin Hammond[1]

Abstract: This paper focuses on the important, but tricky, problem of determining provably correct program properties *automatically* from program source. We describe a novel approach to constructing correct low-level programs. By using modern, full-spectrum *dependent types*, we are able to give an explicit and checkable link between the low-level program and its high-level meaning. Our approach closely links programming and theorem proving in that a type correct program is a constructive proof that the program meets its specification. It goes beyond typical *model-checking* approaches, that are commonly used to check formal properties of low-level programs, by building proofs over abstractions of properties. In this way, we avoid the state-space explosion problem that bedevils model-checking solutions. We are also able to consider properties over potentially infinite domains and determine properties for potentially infinite programs. We illustrate our approach by implementing a carry-ripple adder for binary numbers.

[1]School of Computer Science, University of St Andrews, St Andrews, Scotland; Phone: +44 1334-463253; Email: eb,james,kh@dcs.st-and.ac.uk

1.1 INTRODUCTION

Type theories based on *dependent types* [19, 8] have received significant interest in the functional programming community, promising enhanced mechanisms for expressing correctness properties for functional programs. Where conventional types express a program's meaning, *dependent types* allow types to be predicated on values, *so expressing a more precise meaning*. This allows the programmer both to write programs and to verify specifications within the same framework. *Simply because a program typechecks*, we obtain a *free theorem* [27] that the program conforms to its specification. In this paper, we will show how dependent types allow the relationship to be maintained explicitly between high-level data structures and their concrete representations.

To illustrate our approach we develop a simple program modelling a realistic circuit: a binary *carry-ripple adder*, in which the type of a binary number embeds the decoding function to natural numbers so that the type of binary addition expresses that it must correspond to addition on natural numbers. This program is implemented using IVOR [5], an interactive theorem prover for a *strongly-normalising* dependent-type theory[2], which we call TT, embedded as a Haskell library. The main contributions of this paper are:

- We show how dependent types allow low-level constructions (such as bits, binary numbers and adder circuits) to be given types which express their high level meaning. Low level programs built from these constructions then reflect their high level meaning in their type, giving a program which is *correct by construction* [1], without the need for costly *model-checking* or similar apparatus.
- We give a realistic example of programming with dependent types. The advantage of using dependent types for our example program (or indeed any program where we require strong guarantees) is that dependent types link the proof and the program in a machine checkable manner. Proving correctness by hand is an error prone process; any error in a machine checkable proof will be caught by the typechecker.

Although there has been significant theoretical interest in dependent types, the number of practical examples has been highly limited to date ([17] is a notable exception). By developing such an example in this paper, we hope to demonstrate a wider range of applicability for dependent types than has hitherto been considered, and so to help convince any still-wavering functional programmers of the potential merits of a dependently-typed approach.

Our approach is simlar to *program extraction* [23, 18], or proving a specification in a theorem prover such as COQ [9] or Isabelle [22]. However, our emphasis is different in that we treat the program as the prior notion rather than the proof. By giving a precise type the program *is* the proof of the specification. We are thus able to write clear, concise programs without type annotations (other than the

[2] i.e. one where evaluation of all programs terminates without error.

top level type of each function), and with occasional explicit proof terms where required.

We find this approach greatly assists the construction of correct functional programs. It allows us to concentrate on the program itself and add details of proofs where necessary for well-typedness. helping to achieve the hitherto elusive general promise of ease-of-reasoning that was identified by John Hughes in his seminal 1984 paper "Why Functional Programming Matters" [15].

1.1.1 Programming with Dependent Types

In a dependently-typed language such as TT, we can parameterise types over *values* as well as other types. We say that TT is a *full-spectrum* dependently typed language, meaning that *any* value may appear as part of a type, and types may be computed from any value. For example, we can define a "lists with length" (or vector) type; to do this we first declare a type of natural numbers to represent such lengths:

$$\underline{\text{data}} \quad \mathbb{N} : \star \quad \underline{\text{where}} \quad 0 : \mathbb{N} \quad | \quad \mathsf{s} : \mathbb{N} \to \mathbb{N}$$

This declaration is written using GADT [24] style syntax. It declares three constants, $\mathbb{N}$, 0 and s along with their types, where $\star$ indicates the type of types. Then we may make the following declaration of vectors; note that ε only targets vectors of length zero, and $x::xs$ only targets vectors of length greater than zero:

$$\begin{array}{lll} \underline{\text{data}} & \mathsf{Vect} : \star \to \mathbb{N} \to \star & \underline{\text{where}} \\ & \quad \varepsilon \quad : \mathsf{Vect}\ A\ 0 & \\ \quad | & (::) \quad : A \to (\mathsf{Vect}\ A\ k) \to (\mathsf{Vect}\ A\ (\mathsf{s}\,k)) & \end{array}$$

The type of $\mathsf{Vect} : \star \to \mathbb{N} \to \star$ indicates that it is predicated on a type (the element type) and a natural number (the length). When the type includes explicit length information like this, it follows that a function over that type will express the invariant properties of the length. For example, the type of the program **vPlus**, which adds corresponding numbers in each vector, expresses the invariant that the input vectors are the same length as the output. The program consists of a type declaration (introduced by let) followed by a pattern matching definition:

$$\begin{array}{l} \underline{\text{let}} \quad \mathbf{vPlus} : (\mathsf{Vect}\,\mathbb{N}\,n) \to (\mathsf{Vect}\,\mathbb{N}\,n) \to (\mathsf{Vect}\,\mathbb{N}\,n) \\ \quad \mathbf{vPlus} \quad \varepsilon \quad \varepsilon \quad \mapsto \varepsilon \\ \quad \mathbf{vPlus}\,(x::xs)\,(y::ys) \ \mapsto (x+y)::(\mathbf{vPlus}\,xs\,ys) \end{array}$$

Unlike in a simply typed language, we do not need to give error handling cases when the lengths of the vectors do not match; the typechecker verifies that these cases are impossible.

A key advantage of being able to index types over values is that it allows us to link *representation* with *meaning*. In the above example, the only part of the list's meaning we consider is its length — in the rest of this paper we will consider more detailed examples.

1.1.2 Theorem Proving

The dependent type system of TT also allows us to express properties directly. For example, the following heterogeneous definition of equality, due to McBride [20], is built in to TT:

$$\underline{\text{data}} \quad (=) : (A : \star) \to (B : \star) \to A \to B \to \star \quad \underline{\text{where}}$$
$$\quad \mathsf{refl} : (a : A) \to (a = a)$$

Note that since the range of a function type may depend on previous arguments, it is possible to bind names in the domain, as with refl's domain $a : A$ here. This declaration states that two values in two different types may be equal, but the only way to construct a proof, by reflexivity, is if the two values really *are* equal. For convenience, we use an infix notation for the $=$ type constructor and leave the parameters A and B implicit. Since equality is a datatype just like $\mathbb{N}$ and Vect, we can write programs by pattern matching on instances of equality, such as the following program which can be viewed as a proof that s respects equality:

$$\underline{\text{let}} \quad \mathbf{resp\_s} : (n = m) \to (\mathsf{s}\,n = \mathsf{s}\,m)$$
$$\mathbf{resp\_s}\,(\mathsf{refl}\,n) \;\mapsto\; \mathsf{refl}\,(\mathsf{s}\,n)$$

The type of this function implicitly abstracts over all n and m. It is implemented by pattern matching on the first argument, of type $n = m$. The only way to construct such a value is from $\mathsf{refl}\,n : n = n$, therefore to complete the definition it is sufficient to construct a proof of $\mathsf{s}\,n = \mathsf{s}\,n$.

We can use the equality type to perform equational reasoning on programs — a term of type $P\,x$ can be transformed to $P\,y$ if we have a proof of $x = y$, using the following function:

$$\underline{\text{let}} \quad \mathbf{repl} : (x : A) \to (y : A) \to (x = y) \to (P : A \to \star) \to P\,x \to P\,y$$
$$\mathbf{repl}\,x\,y\,(\mathsf{refl}\,x)\,P\,p \;\mapsto\; p$$

Using this function (and an interactive theorem prover such as IVOR to assist in applying it correctly), we can build explicit and machine checkable proofs of any desired property, e.g. commutativity and associativity of addition.

In this paper, we will implement a carry-ripple adder for binary numbers, using inductive families to maintain a strong link between binary numbers and the natural numbers they represent. We will use these theorem proving techniques to construct proofs that addition of binary numbers corresponds to addition of unary natural numbers.

1.2 BINARY NUMBER REPRESENTATION

In this section we develop some primitive constructions which we will use to build a correct by construction binary adder. *Correct by construction* means that the fact that the program typechecks is a guarantee that it conforms to the specification given in the type. We use the word "correct" in a very strong sense; the totality

of the type theory guarantees that a program will yield a result conforming to the specification in *all* cases.

1.2.1 Natural number arithmetic

We predicate binary numbers on their decoding as a $\mathbb{N}$, which means that binary addition must correspond to natural number addition. Addition is defined inductively, as an infix function $+$, as follows:

$$\begin{array}{ll}\underline{\text{let}} & (+) : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ \mathsf{0} & + y \mapsto y \\ (\mathsf{s}\, k) & + y \mapsto \mathsf{s}\,(k + y)\end{array}$$

Multiplication can be defined similarly. It is straightforward to show several properties of these definitions in TT itself using IVOR's rewriting tactic (implemented via the **repl** function in section 1.1.2), e.g. the following are functions giving proofs of commutativity and associativity respectively:

$$\begin{array}{l}\textbf{comm\_plus} : (x : \mathbb{N}) \to (y : \mathbb{N}) \to (x + y = y + x) \\ \textbf{assoc\_plus} : (x : \mathbb{N}) \to (y : \mathbb{N}) \to (z : \mathbb{N}) \to ((x + y) + z = x + (y + z))\end{array}$$

By proving these properties, and showing that binary addition is equivalent to natural number addition by indexing binary numbers by the corresponding $\mathbb{N}$, we get these properties for free on binary numbers.

We briefly illustrate theorem proving in dependent type theory by showing a proof term for **comm_plus** above. Let us assume the following two lemmas (both simple to prove):

$$\begin{array}{l}\textbf{plus\_0} : (n : \mathbb{N}) \to (n = n + \mathsf{0}) \\ \textbf{plus\_s} : (n, m : \mathbb{N}) \to (\mathsf{s}\,(m + n) = m + (\mathsf{s}\, n))\end{array}$$

The definition of plus gives rise to reduction rules which can be exploited by the typechecker; it is defined by pattern matching on the first argument, so an expression $a + b$ can always be reduced if a is constructor headed. The above lemmas give rewrite rules for the situation where b is in constructor form by a is not. Then **comm_plus** can be written as a recursive pattern matching function, making use of the **repl** function for equational reasoning:

$$\begin{array}{lll}\textbf{comm\_plus} & \mathsf{0} \quad m & \mapsto \textbf{plus\_0}\, m \\ \textbf{comm\_plus} & (\mathsf{s}\, k)\, m & \mapsto \textbf{repl}\,(\mathsf{s}\,(m + k))\,(m + (\mathsf{s}\, k))\,(\textbf{plus\_s}\, m\, k) \\ & & \qquad (\lambda a : \mathbb{N}.\, \mathsf{s}\,(k + m) = a) \\ & & \qquad (\textbf{resp\_s}\,(\textbf{comm\_plus}\, k\, m))\end{array}$$

In the recursive case, **repl** is used to rewrite the type — it changes the type from $\mathsf{s}\,(k + m) = m + (\mathsf{s}\, k)$ to $\mathsf{s}\,(k + m) = \mathsf{s}\,(m + k)$. Then we can apply the function recursively (we think of this as applying the induction hypothesis) to the proof that equality respects successor given in section 1.1.2.

In practice, since the typechecker can construct several arguments to **repl** automatically, since it knows the types we are rewriting between, we will write

applications of **repl** in the following way, eliding all but the rewriting lemma to be applied and the value to be returned:

$$\begin{aligned}&\mathbf{comm\_plus}\ \ \mathsf{0}\ \ m\ \ \mapsto \mathbf{plus\_0}\, m\\ &\mathbf{comm\_plus}\,(\mathsf{s}\,k)\, m\ \ \mapsto \mathbf{repl}\,(\mathbf{plus\_s}\, m\, k)\,(\mathbf{resp\_s}\,(\mathbf{comm\_plus}\, k\, m))\end{aligned}$$

1.2.2 Bit representation

The simplest building block of a binary number is the bit; in a traditional simply typed programming language we might represent this as a boolean. However, since we are interested in preserving meaning throughout the whole computation, we express in the type that bits on their own represent either zero or one:

$$\underline{\text{data}}\quad \mathsf{Bit}\ :\ \mathbb{N}\to\star\quad \underline{\text{where}}\quad \mathsf{O}\ :\ \mathsf{Bit}\,\mathsf{0}\quad |\quad \mathsf{I}\ :\ \mathsf{Bit}\,\mathsf{1}$$

For readability we will use traditional notation for numbers and arithmetic operators, except in pattern matching definitions; e.g. we write $\mathsf{I}\ :\ \mathsf{Bit}\,\mathsf{1}$ rather than $\mathsf{I}\ :\ \mathsf{Bit}\,(\mathsf{s}\,\mathsf{0})$.

We will also need to add pairs of bits, which will result in a two bit number. It is convenient to represent bit pairs in their own type, with the corresponding decoding:

$$\begin{aligned}&\underline{\text{data}}\quad \mathsf{BitPair}\ :\ \mathbb{N}\to\star\quad \underline{\text{where}}\\ &\quad \mathsf{bitpair}\ :\ \mathsf{Bit}\,b\ \to\ \mathsf{Bit}\,c\ \to\ \mathsf{BitPair}\,(2\times b+c)\end{aligned}$$

1.2.3 A full adder

Various algorithms can be used to adding n-bit binary numbers in hardware. Whichever we choose, the required primitive operation is a **full adder**. A full adder is a circuit which adds three binary digits, producing two digits (a sum, and the carry value). Three inputs, l, r and c_{in} are combined into two outputs, s and c_{out}:

$$\begin{aligned}s &= (l\,\mathsf{xor}\,r)\,\mathsf{xor}\,c_{in}\\ c_{out} &= (l\,\mathsf{and}\,r)\,\mathsf{or}\,(r\,\mathsf{and}\,c_{in})\,\mathsf{or}\,(c_{in}\,\mathsf{and}\,l)\end{aligned}$$

It is well known that this is a correct model of a full adder. It is straightforward to check by hand by constructing a truth table for the three inputs l, r and c_{in}. Using dependent types, we can let the machine check that we have a correct implementation with respect to the desired behaviour. In this section we give two implementations of the full adder: firstly, a simple definition by pattern matching; and secondly, a definition implementing the above calculation through logic gates.

Simple definition

We model this circuit with the following function, **addBit**, with the type guaranteeing that the resulting bit representation decodes to the sum of the inputs. Our

definition is simply a lookup table enumerating all of the possibilities; typechecking of this is fully automatic since the types of the left and right hand side of each equation are identical and any reduction of $+$ required at the type level is on constructor headed arguments.

$$
\begin{array}{ll}
\underline{\text{let}} & \textbf{addBit} : \mathsf{Bit}\,c \to \mathsf{Bit}\,x \to \mathsf{Bit}\,y \to \mathsf{BitPair}\,(c+x+y) \\
& \textbf{addBit}\,\mathsf{O}\,\mathsf{O}\,\mathsf{O} \mapsto \mathsf{bitpair}\,\mathsf{O}\,\mathsf{O} \\
& \textbf{addBit}\,\mathsf{O}\,\mathsf{O}\,\mathsf{I} \mapsto \mathsf{bitpair}\,\mathsf{O}\,\mathsf{I} \\
& \textbf{addBit}\,\mathsf{O}\,\mathsf{I}\,\mathsf{O} \mapsto \mathsf{bitpair}\,\mathsf{O}\,\mathsf{I} \\
& \textbf{addBit}\,\mathsf{O}\,\mathsf{I}\,\mathsf{I} \mapsto \mathsf{bitpair}\,\mathsf{I}\,\mathsf{O} \\
& \textbf{addBit}\,\mathsf{I}\,\mathsf{O}\,\mathsf{O} \mapsto \mathsf{bitpair}\,\mathsf{O}\,\mathsf{I} \\
& \textbf{addBit}\,\mathsf{I}\,\mathsf{O}\,\mathsf{I} \mapsto \mathsf{bitpair}\,\mathsf{I}\,\mathsf{O} \\
& \textbf{addBit}\,\mathsf{I}\,\mathsf{I}\,\mathsf{O} \mapsto \mathsf{bitpair}\,\mathsf{I}\,\mathsf{O} \\
& \textbf{addBit}\,\mathsf{I}\,\mathsf{I}\,\mathsf{I} \mapsto \mathsf{bitpair}\,\mathsf{I}\,\mathsf{I}
\end{array}
$$

Lower level operations

Any definition of **addBit** which satisfies the above type is guaranteed by construction to be a correct implementation. We can therefore model a full adder by modelling the required hardware operations (and, or and xor), then combining them to produce a full adder, i.e.:

$$
\begin{array}{rl}
\textbf{andGate} & : \mathsf{Bit}\,x \to \mathsf{Bit}\,y \to \mathsf{Bit}\,(\textbf{and}\,x\,y) \\
\textbf{orGate} & : \mathsf{Bit}\,x \to \mathsf{Bit}\,y \to \mathsf{Bit}\,(\textbf{or}\,x\,y) \\
\textbf{xorGate} & : \mathsf{Bit}\,x \to \mathsf{Bit}\,y \to \mathsf{Bit}\,(\textbf{xor}\,x\,y)
\end{array}
$$

Since bits are indexed over their interpretation as natural numbers, we need to provide the meaning of each of these operations on natural numbers as well as the operation itself. We treat zero as false, and non-zero as true. For example, to model an and gate, we first define the corresponding function over $\mathbb{N}$:

$$
\begin{array}{ll}
\underline{\text{let}} & \textbf{and} : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
& \textbf{and}\,(\mathsf{s}\,x)\,(\mathsf{s}\,y) \mapsto 1 \\
& \textbf{and}\;\;0\;\;\;y \mapsto 0 \\
& \textbf{and}\;\;x\;\;\;0 \mapsto 0
\end{array}
$$

Then an and gate modelling the behaviour of Bits can be implemented:

$$
\begin{array}{ll}
& \textbf{andGate}\,\mathsf{I}\,\mathsf{I} \mapsto \mathsf{I} \\
& \textbf{andGate}\,\mathsf{O}\,y \mapsto \mathsf{O} \\
& \textbf{andGate}\,x\,\mathsf{O} \mapsto \mathsf{O}
\end{array}
$$

The or and xor operations are implemented similarly. With these definitions, we can implement **addBit** as a combination of logic gates, given the earlier definitions of the sum and carry outputs:

```
let   addBit : Bit c → Bit x → Bit y → BitPair (c + x + y)
  addBit l r c_in
    ↦ let c_out = orGate (orGate (andGate l r) (andGate r c_in)) (andGate c_in l)
          s = xorGate (xorGate l r) c_in
        in bitpair c_out s
```

We can be sure this is a correct implementation, because the typechecker verifies that the *meaning* of the pair of output bits is correct with respect to the meaning of the inputs. A small amount of theorem proving is required to help the type checker; case analysis on each input c, x and y verifies that the types of the left and right hand sides are always equal. We read the type as a specification that, given bits with numeric meanings c, x and y, the output pair has numeric meaning $c+x+y$.

1.2.4 Number representation

There are two possibilities for a binary representation built on Bits — either left or right-biased. A number is a list of bits; the choice determines whether a new bit value b is added at the most significant end (changing a value n to $2^{width} \times b + n$), or at the least significant end (changing a value n to $2 \times n + b$). As TT types, the choices are:

1. Left-biased; adding most significant bit:

```
data   Number : ℕ → ℕ → ⋆   where
         none : Number 0 0
       | bit  : Bit b → Number width val →
                  Number (1 + width) (2^width × b + val)
```

2. Right-biased; adding least significant bit:

```
data   NumberR : ℕ → ℕ → ⋆   where
         noneR : NumberR 0 0
       | bitR  : NumberR width val → Bit b →
                  NumberR (1 + width) (2 × val + b)
```

Which we choose has important consequences. Not only does it affect how the binary adder is written (and which algorithms are simplest to implement), but also how we show the correspondence with addition on $\mathbb{N}$.

Number is indexed over its width (i.e. the number of bits) as well as its decoding. As well as helping to compute the decoding, this allows us to specify the effect an operation has on a number's width — e.g. addition of two n-bit numbers gives an n-bit number with carry, multiplication of two n-bit numbers gives a $2 \times n$-bit number.

1.2.5 Number equivalence

The representation we choose, Number or NumberR has a significant effect on how we show equivalance between binary and unary functions. Each models a list of bits, but we may find some functions easier to construct in one setting than another. In fact, we can show that the two representations are equivalent and hence interchangeable using the fact that we have indexed the binary representation over its decoding as a $\mathbb{N}$. It suffices to implement the following functions:

$$\underline{\text{let}} \quad \textbf{leftToRight} \ : \ \mathsf{Number}\, w\, val \rightarrow \mathsf{NumberR}\, w\, val$$
$$\underline{\text{let}} \quad \textbf{rightToLeft} \ : \ \mathsf{NumberR}\, w\, val \rightarrow \mathsf{Number}\, w\, val$$

We do not need to show that the composition of these two functions is the identity function. The property in which we are interested is that translating between representations preserves the decoding. Since the type expresses that decoding is preserved, we know that composition of **leftToRight** and **rightToLeft** gives a binary number with the same meaning as the original.

To illustrate this, we will show how to implement **leftToRight**. The simplest method in general is to provide functions to construct a NumberR with the same indices as the constructors for Number:

$$\textbf{nonerL} \ : \ \mathsf{NumberR}\, 0\, 0$$
$$\textbf{bitrL} \ : \ \mathsf{Bit}\, bv \rightarrow \mathsf{NumberR}\, w\, val \rightarrow \mathsf{NumberR}\, (1+w)\, (2^w \times bv + val)$$

Implementing **nonerL** is straightforward, using the constructor noneR. For **bitrL**, we must show how to add a most significant bit to a right-biased number, where the bitR constructor adds the least significant bit. We might expect the following definition to work, simply pushing the bit to be added, b, through recursive calls:

$$\begin{array}{ll} \underline{\text{let}} \quad \textbf{bitrL} \ : \ \mathsf{Bit}\, bv \rightarrow \mathsf{NumberR}\, w\, val \rightarrow \mathsf{NumberR}\, (1+w)\, (2^w \times bv + val) & \\ \quad \textbf{bitrL}\, b \quad \mathsf{noneR} & \mapsto \mathsf{bitR}\, \mathsf{noneR}\, b \\ \quad \textbf{bitrL}\, b\, (\mathsf{bitR}\, n\, b_l) & \mapsto \mathsf{bitR}\, (\textbf{bitrL}\, b\, n)\, b_l \end{array}$$

However, as it stands, this does not typecheck. We need to do a little extra manipulation of the types, since the type of the bitR constructor does not reflect the fact that the new bit is added as the high bit. The expected return type of the recursive case is:

$$\mathsf{NumberR}\, (2+w)\, (2^{1+w} \times bv_l + (2 \times nv + bv))$$

nv is the decoding of n, and bv_l the decoding of b_l. However, the type we get from the construction with bitR is:

$$\mathsf{NumberR}\, (2+w)\, (2 \times (2^w \times bv_l + nv) + bv)$$

It is easy to see, by simple algebraic manipulation, that the two expressions are equivalent. In order to convince the typechecker, however, we need an extra lemma which converts the return type we have into the return type we need for the value we wish to construct:

$$\textbf{bitrL\_lemma} \ : \ (2^{1+w} \times bv_l + (2 \times nv + bv)) = (2 \times (2^w \times bv_l + nv) + bv)$$

This lemma can be applied using **repl**; the correct definition of **bitrL** is:

$$\begin{array}{l}\underline{\text{let}}\quad \mathbf{bitrL} : \mathsf{Bit}\,bv \to \mathsf{NumberR}\,w\,val \to \mathsf{NumberR}\,(1+w)\,(2^w \times bv + val)\\ \quad \mathbf{bitrL}\,b \quad \mathsf{noneR} \quad \mapsto \mathsf{bitR}\,\mathsf{noneR}\,b\\ \quad \mathbf{bitrL}\,b\,(\mathsf{bitR}\,n\,b_l) \;\mapsto \mathbf{repl}\,\mathbf{bitrL\_lemma}\,(\mathsf{bitR}\,(\mathbf{bitrL}\,b\,n)\,b_l)\end{array}$$

Implementing **bitrL_lemma** is also through equational reasoning with **repl**. IVOR has a library of useful theorems about addition and multiplication to assist with this; such a function could in many cases be constructed via the Omega decision procedure [25]. Having implemented **nonerL** and **bitrL**, the definition of **leftToRight** is a straightforward application of these functions.

$$\begin{array}{l}\underline{\text{let}}\quad \mathbf{leftToRight} : \mathsf{Number}\,w\,val \to \mathsf{NumberR}\,w\,val\\ \quad \mathbf{leftToRight}\;\;\mathsf{none} \quad \mapsto \mathbf{nonerL}\\ \quad \mathbf{leftToRight}\,(\mathsf{bit}\,b\,n) \;\mapsto \mathbf{bitrL}\,b\,n\end{array}$$

The implementation of **rightToLeft** proceeds similarly, pushing the bit to be added through recursive calls and rewriting types through equational reasoning where required. Rewriting types in this way is a common pattern when implementing functions indexed by arithmetic operations, and it is therefore vital to have the support of a theorem proving environment in which the types can *direct* the implementation.

Having implemented these definitions, we are free to choose either representation for any function, and in practice we are likely to choose the representation which yields the more straightforward proofs.

For our adder, we choose the left-biased representation. Although it looks slightly more complex, in that the value depends on the number's width as well as the bit value, its main practical advantage is that it leads to a slightly simpler definition of full binary addition, with correspondingly simpler algebraic manipulations in order to prove correspondence with $\mathbb{N}$ addition.

1.2.6 Carry

The final part of the representation pairs a number with a carry bit. In order to deal with overflow, our addition function over n-bit numbers will give an n-bit number and a carry bit. This also means that n-bit add with carry can be implemented recursively in terms of $n-1$-bit add with carry, with easy access to the carry bit resulting from the recursive call.

$$\begin{array}{l}\underline{\text{data}}\quad \mathsf{NumCarry} : \mathbb{N} \to \mathbb{N} \to \star \quad \underline{\text{where}}\\ \quad \mathsf{numcarry} : \mathsf{Bit}\,c \to \mathsf{Number}\,width\,val \to\\ \qquad\qquad\qquad \mathsf{NumCarry}\,width\,(2^{width} \times c + val)\end{array}$$

1.3 A CARRY RIPPLE ADDER

We can now define an arbitrary width binary addition with carry function, **addNumber**. We can choose between several implementations of this, e.g. carry-ripple, or carry

lookahead. However, because the type of the function precisely expresses its *meaning* (namely, that it implements addition on binary numbers corresponding to natural number addition), we can be sure that any type-correct implementation is equivalent to any other. The type of **addNumber** is as follows:

$$\begin{array}{ll}\underline{\text{let}} & \textbf{addNumber} \;:\; \mathsf{Number}\,\mathit{width}\,x \to \mathsf{Number}\,\mathit{width}\,y \to \mathsf{Bit}\,c \to \\ & \qquad \mathsf{NumCarry}\,\mathit{width}\,(x+y+c)\end{array}$$

1.3.1 First attempt

We will define a carry ripple adder. This is a simple algorithm — to add two n-bit numbers, first add the least significant ($n-1$-bit) numbers, then add the most significant bits with the carry resulting from the recursive addition. We would like to define **addNumber** as follows:

$$\begin{array}{l}\textbf{addNumber}\quad \mathsf{none}\qquad \mathsf{none}\quad \mathit{carry} \;\mapsto\; \mathsf{numcarry}\,c\,\mathsf{none} \\ \textbf{addNumber}\,(\mathsf{bit}\,\mathit{bx}\,\mathit{nx})\,(\mathsf{bit}\,\mathit{by}\,\mathit{ny})\,\mathit{carry} \\ \qquad\qquad \mapsto\; \underline{\text{let}}\,\mathsf{numcarry}\,\mathit{carry}_0\,\mathit{rec} \;=\; \textbf{addNumber}\,\mathit{nx}\,\mathit{ny}\,\mathit{carry} \\ \qquad\qquad\quad \underline{\text{let}}\,\mathsf{bitpair}\,\mathit{carry}_1\,s \;=\; \textbf{addBit}\,\mathit{bx}\,\mathit{by}\,\mathit{carry}_0 \\ \qquad\qquad\qquad \underline{\text{in}}\,\mathsf{numcarry}\,\mathit{carry}_1\,(\mathsf{bit}\,s\,\mathit{rec})\end{array}$$

Although this *looks* like a correct implementation of add with carry, unfortunately it does not typecheck as written. The problem is similar to that encountered in our definition of **leftToRight** in Section 1.2.5, but more complex. Let us examine the types of the left- and right-1hand sides of the recursive case. We have:

$$\begin{array}{l}\mathit{bx} \;:\; \mathsf{Bit}\,\mathit{bxv} \\ \mathit{by} \;:\; \mathsf{Bit}\,\mathit{byv} \\ \mathit{nx} \;:\; \mathsf{Number}\,w\,\mathit{nxv} \\ \mathit{ny} \;:\; \mathsf{Number}\,w\,\mathit{nyv} \\ \mathit{carry} \;:\; \mathsf{Bit}\,c \\ \mathsf{bit}\,\mathit{bx}\,\mathit{nx} \;:\; \mathsf{Number}\,(1+w)\,(2^w \times \mathit{bxv} + \mathit{nxv}) \\ \mathsf{bit}\,\mathit{by}\,\mathit{ny} \;:\; \mathsf{Number}\,(1+w)\,(2^w \times \mathit{byv} + \mathit{nyv})\end{array}$$

Here, *bxv*, *byv*, *nxv* and *nyv* are the decodings of the bits and numbers, and w is the width of the numbers in the recursive call. The type of the left hand side is:

$$\begin{array}{l}\textbf{addNumber}\,(\mathsf{bit}\,\mathit{bx}\,\mathit{nx})\,(\mathsf{bit}\,\mathit{by}\,\mathit{ny})\,\mathit{carry} \\ \qquad\qquad :\; \mathsf{NumCarry}\,(1+w)\,((2^w \times \mathit{bxv} + \mathit{nxv}) + (2^w \times \mathit{byv} + \mathit{nyv}) + c)\end{array}$$

Typechecking the right hand side proceeds as follows. We first check the type of the recursive call:

$$\textbf{addNumber}\,\mathit{nx}\,\mathit{ny}\,\mathit{carry} \;:\; \mathsf{NumCarry}\,w\,(\mathit{nxv} + \mathit{nyv} + c)$$

From the recursive call, we obtain the sum of the lower bits and a carry flag by pattern matching the result against $\mathsf{numcarry}\,\mathit{carry}_0\,\mathit{rec}$. Assuming that carry_0 has type $\mathsf{Bit}\,c_0$ and *rec* has type $\mathsf{Number}\,w\,\mathit{nrec}$, we get:

$$\mathsf{numcarry}\,\mathit{carry}_0\,\mathit{rec} \;:\; \mathsf{NumCarry}\,w\,(2^w \times c_0 + \mathit{nrec})$$

The decodings c_0 and *rec* are introduced by the pattern matching against numcarry, and we can infer that $(2^w \times c_0 + nrec) = (nxv + nyv + c)$. Given this, we can check the type of adding the top bits:

$$\textbf{addBit}\, bx\, by\, carry_0 \;:\; \mathsf{BitPair}\,(bxv + byv + c_0)$$
$$\mathsf{bitpair}\, carry_1\, s \;:\; \mathsf{BitPair}\,(2 \times ns + c_1)$$

Again, *ns* and c_1 are introduced by pattern matching and we can infer that $(2 \times ns + c_1) = (bxv + byv + c_0)$. Finally, the result of the function is checked as follows:

$$\mathsf{numcarry}\, carry_1\, (\mathsf{bit}\, s\, rec) \;:\; \mathsf{NumCarry}\,(1+w)\,(2^{1+w} \times c_1 + (2^w \times ns + nrec))$$

For typechecking to succeed, we need to know that $(2^w \times c_1 + (2^w \times ns + nrec))$ and $((2^w \times bxv + nxv) + (2^w \times byv + nyv) + c)$ are convertible, in order to unify the type of the left hand side with the type of the right hand side. Unfortunately, we cannot expect this to happen automatically — it requires the typechecker to do some automatic equational reasoning given the equalities inferred by the pattern matching let bindings.

1.3.2 A correct carry ripple adder

Although our definition appears correct, the type of the second clause does not quite work automatically. We need to provide an explanation, in the form of a checkable proof using equational reasoning with **repl** as described in Section 1.1.2, that the values in the left- and right-hand sides of the pattern clause can be converted to each other. We write a helper function which makes the dependencies explicit in its type. We will write **addNumber** as follows:

$$\begin{aligned}&\textbf{addNumber}\quad \mathsf{none}\qquad \mathsf{none}\quad carry \;\mapsto\; \mathsf{numcarry}\, c\, \mathsf{none}\\ &\textbf{addNumber}\,(\mathsf{bit}\; bx\; nx)\,(\mathsf{bit}\; by\; ny)\; carry\\ &\qquad\qquad \mapsto\; \textbf{bitCase}\; carry\; bx\; by\; nx\; ny\; (\textbf{addNumber}\; nx\; ny\; carry)\end{aligned}$$

The purpose of **bitCase** is to convert the type we want to give for the right hand side to the type required by the typechecker. In practice, this is where we do the theorem proving required to show the correctness of binary addition.

$$\begin{aligned}\underline{\mathsf{let}}\quad \textbf{bitCase} :\; &\mathsf{Bit}\, c \to \mathsf{Bit}\; bxv \to \mathsf{Bit}\; byv \to\\ &\mathsf{Number}\; w\; nxv \to \mathsf{Number}\; w\; nyv \to\\ &\mathsf{NumCarry}\; w\,(nxv + nyv + c) \to\\ &\mathsf{NumCarry}\;(1+w)\,((2^w \times bxv + nxv) + (2^w \times byv + nyv) + c)\end{aligned}$$

The advantage of writing the definition in this way, passing the result of the recursive call to **addNumber** in to **bitCase**, is that the dependencies between the decodings of the numbers are maintained. To write **bitCase**, we follow a method similar to that for writing **bitlR** in Section 1.2.5; i.e. separate the construction of the data and the rewriting of the type through equational reasoning. Constructing the data involves adding the most significant bits, and appending the resulting bit pair to the result of the recursive call. We implement this with the following helper:

$$\begin{array}{ll}\underline{\text{let}} & \textbf{bitCase}' \ : \ \mathsf{BitPair}\, bv \to \mathsf{Number}\, w\, val \to \\ & \qquad \mathsf{NumCarry}\,(w+1)\,(2^w \times bv + val) \\ \multicolumn{2}{l}{\textbf{bitCase}'\,(\mathsf{bitpair}\, b\, c)\, num \ \mapsto \ \textbf{repl}\, \textbf{bitCase}'\textbf{p}\,(\mathsf{numcarry}\, b\,(\mathsf{bit}\, c\, num))}\end{array}$$

We use the **bitCase′p** lemma to rewrite the type so that the numcarry constructor can be applied. **bitCase′p** has the following type, as is easy to show by equational reasoning:

$$\textbf{bitCase}'\textbf{p} \ : \ (2^w \times (val + 2 \times cv) + bv) = (2^{w+1} \times cv + (2^w \times val + bv))$$

Finally, we observe that **bitCase** is easier to write if we rewrite the type so that the argument and return type include the common subterms $nxv + nyv + c$, since we can then treat this subterm as atomic. We also lift the common subterm 2^w, to give:

$$\begin{array}{ll}\underline{\text{let}} & \textbf{bitCase2} \ : \ \mathsf{Bit}\, c \to \mathsf{Bit}\, bxv \to \mathsf{Bit}\, byv \to \\ & \qquad \mathsf{Number}\, w\, nxv \to \mathsf{Number}\, w\, nyv \to \\ & \qquad \mathsf{NumCarry}\, w\,(nxv + nyv + c) \to \\ & \qquad \mathsf{NumCarry}\,(1+w)\,(2^w \times (bxv + byv) + (nxv + nyv + c))\end{array}$$

The types are now in a form where the helper **bitCase′** can be applied directly:

$$\begin{array}{l}\textbf{bitCase2}\, carry\, bx\, by\, nx\, ny\,(\mathsf{numcarry}\, carry_0\, val) \\ \qquad \mapsto \ (\textbf{bitCase}'\,(\textbf{addBit}\, bx\, by\, carry_0)\, val)\end{array}$$

bitCase itself is written by rewriting the type to be in a form usable by **bitCase2**, using a suitable lemma **bitCaseRewrite**:

$$\begin{array}{l}\textbf{bitCase}\, carry\, bx\, by\, nx\, ny\, val \\ \qquad \mapsto \ \textbf{repl}\, \textbf{bitCaseRewrite}(\textbf{bitCase2}\, carry\, bx\, by\, nx\, ny)\end{array}$$

The **bitCaseRewrite** lemma simply expresses the equality between the indices in the types of **bitCase** and **bitCase2** and allows conversion between the two:

$$\begin{array}{l}\textbf{bitCaseRewrite} \ : \ ((2^w \times bxv + nxv) + (2^w \times byv + nyv) + c) = \\ \qquad\qquad (2^w \times (bxv + byv) + (nxv + nyv + c))\end{array}$$

This function has required us to break down the construction of the number into its component parts: adding the upper bits; making a recursive call; and gluing the results together. Each step has required some fairly simple equational reasoning to convert the decoding of the numbers and bits into the required type; the required lemmas can be implemented by a series of rewrites using **repl**. The full development, as an IVOR script constructing a TT program, is available online[3].

Since TT is a language of total functions, we can be confident that the above definition will always terminate. This is an important consideration, since part of guaranteeing the correctness of a function is the guarantee that it will yield a result for *all* type-correct inputs, not just a subset.

Although extra work is needed at the type level to show that the decodings of the binary numbers are consistent in the implementation of **bitCase**, the *computational* content is the same as our first attempt at defining **addNumber**. The

[3] http://www.dcs.st-and.ac.uk/%7Eeb/CarryRipple

decoding and length data appear only at the type level, so need not be stored at run-time [7]. As a result, any function such as **repl** which merely manipulates the indices of a type can be replaced by the identity function at run-time, as shown in [4]. What remains is the computational content, i.e. just the bit manipulation.

1.3.3 Properties of addNumber

Since binary numbers are indexed over the natural numbers, and **addNumber** is correspondingly indexed over natural number addition, we should expect it to have corresponding properties. However, we do not have to prove these properties separately, but rather make use of properties we have already proved for $\mathbb{N}$. To do this, we make use of the following lemma:

$$\begin{array}{ll} \underline{\text{let}} & \textbf{numCarryUnique} : (x : \mathsf{NumCarry}\, w\, val) \to \\ & \qquad\qquad\qquad\quad (y : \mathsf{NumCarry}\, w\, val) \to (x = y) \end{array}$$

This lemma states that any two representations which decode to the same $\mathbb{N}$ are equal, and is implemented by structural decomposition of x and y — it is clear, looking at the indices of each constructor, that at each stage only one representation is possible.

Then the proof of commutativity of **addNumber** (setting the carry bit to zero for simplicity) is written as follows, using a lemma to expose the natural number addition in the type and rewriting it with **comm_plus**:

$$\begin{array}{ll} \underline{\text{let}} & \textbf{commAddNumberAux} : (x : \mathsf{NumCarry}\, w\, (lv + rv) + 0) \to \\ & \qquad\qquad\qquad\qquad\quad (y : \mathsf{NumCarry}\, w\, (rv + lv) + 0) \to \\ & \qquad\qquad\qquad\qquad\quad (x = y) \\ \multicolumn{2}{l}{\textbf{commAddNumberAux}\, x\, y} \\ \multicolumn{2}{l}{\quad \mapsto \textbf{repl}\, (\textbf{comm\_plus}\, lv\, rv)\, (\textbf{numCarryUnique}\, x\, y)} \end{array}$$

This function rewrites the type of x of using **comm_plus** to swap lv and rv, then uses **numCarryUnique** to show that the numbers x and y must be equal because their decodings are equal. It is possible to apply **numCarryUnique** only after rewriting the type of x with **comm_plus** so that it is the same as the type of y. We finish the proof of commutativity by applying the above lemma:

$$\begin{array}{ll} \underline{\text{let}} & \textbf{commAddNumber} : (l : \mathsf{Number}\, w\, lv) \to (r : \mathsf{Number}\, w\, rv) \to \\ & \qquad\qquad\qquad\quad (\textbf{addNumber}\, l\, r\, \mathsf{O} = \textbf{addNumber}\, r\, l\, \mathsf{O}) \\ \multicolumn{2}{l}{\textbf{commAddNumber}\, l\, r \;\mapsto \textbf{commAddNumberAux}\, (\textbf{addNumber}\, l\, r\, \mathsf{O})} \\ & \qquad\qquad\qquad\qquad\qquad\qquad\quad (\textbf{addNumber}\, r\, l\, \mathsf{O}) \end{array}$$

We have shown that **addNumber** is commutative without having to look at its definition at all — just using the properties of the function which gives its meaning. Indeed, *any* implementation of **addNumber** with the same type can be substituted into this proof. The main difficulty is rewriting the types to a form to which the natural number proofs can be applied.

1.3.4 Incremental development

Writing a program such as **addNumber** requires us to be aware of the required type of each subterm, and the type of each operation we wish to apply. As we can see by examining the implemention of **bitCase** and its helper operations, such manipulation is difficult to manage by hand, even for relatively simple properties such as the correctness of **addNumber**. We therefore consider it essential that a practical dependently typed language allows the programmer to develop a program interactively (as with EPIGRAM [21] or theorem proving systems such as COQ [9]), rather than the traditional approach of submitting a monolithic program to a compiler.

1.4 RELATED WORK

The most closely related approach of which we are aware is the reFL^{ect} [11] language for verifying hardware circuit designs. Like reFL^{ect}, we are able to verify programs in the language TT itself. Unlike reFL^{ect}, however, we do not implement a theorem prover in TT, but rather use an interactive development system (IVOR) to construct well typed TT programs. The soundness of the system as a whole then relies solely on the soundness of the TT typechecker. This is a key advantage of our approach, since we rely only on the correctness of a checker for a standard and well understood type theory (similar to COQ [9] or EPIGRAM's ETT [8]) rather than external software.

Herrmann's approach [14] is also related to our own in that he constructs high level combinators to generate very specific low level code. He uses meta-programming (in Template Haskell) to generate type and size correct circuit designs, where size errors are caught by the Haskell type checker. We have previously studied correct hardware implementation in HW-Hume [12], which uses high level functional notation to implement low level concepts, but requires external tools (e.g. a model checker) to prove correctness properties. A closely related approach, similarly separating high level description from the low level code, is to use multi-stage programming [10]. A multi-stage language allows specialisation of high level generic abstractions to specific instances, preserving type information between stages. A similar approach has been applied to generate high-performace parallel programs [13]. We are extending our approach with dependently typed multi-stage programming [6] — since the types encode correctness properties, specialisation of a dependently typed program preserves these correctness properties, and we hope to adapt this method to generate hardware descriptions, e.g. via Lava [3], a Haskell-embedded domain specific language for describing circuits.

1.5 CONCLUSION

We have described an approach to constructing correct software with dependent types, and given a specific application area which can benefit from this approach,

namely modelling of hardware circuits. The example we have presented — a binary adder — is relatively straightforward, but illustrates the important concepts behind our approach; namely that each data structure is explicitly linked with its high level meaning (in the case of binary numbers, their decoding as a $\mathbb{N}$). Writing the program proceeds largely as normal, with the additional requirement that we insert rewriting lemmas to preserve well-typedness, and with the benefit that any properties of the original, simpler definition are properties of the low level definition for free, such as our commutativity proof **commAddNumber**. The need to insert rewriting lemmas to show that our definition respects the original natural number implementation is the main cost of our approach. We believe this to be a small price to pay for guaranteed correctness. A part of our approach which we therefore consider *vital* to its practicality is the use of *type-directed* program development — when writing **bitCase**, for example, it is convenient to be able to see the required type for the right hand side of a pattern clause and do the equational reasoning *interactively*.

An alternative approach may be to use Haskell with Generalised Algebraic Data Types [24] (GADTs). GADTs allow limited properties of data types to be expressed by reflecting values at the type level. However, the kind of equational reasoning we need to implement **bitCase** is likely to be very difficult. Other weaker dependent type systems, such as sized types [16] or DML [28] allow types to be parametrised over numbers, but require an external constraint solver which does not allow the user directed equational reasoning required for **bitCase**.

Circuits are usually designed with respect to three views; behaviour, structure and geometry. In a circuit description language such as VHDL or Haskell embedded domain specific language for circuit design such as Lava [3], it is possible to describe these views. Here, we have described the *behavioural* view and shown it to be correct through the type system. We hope to apply multi-stage programming with dependent types [26, 6] to transform programs between these views and generate hardware descriptions from high level programs such as **addNumber**.

The method could be used to verify more complex operations on a CPU, for example multiplication, or in conjunction with program generation techniques and a hardware description language such as Lava or Wired [2] to develop correct by construction FPGA configurations In future, we plan to explore multi-stage programming techniques [10, 13, 6], possibly in combination with Lava, Wired or HW-Hume, to generate correct code.

ACKNOWLEDGEMENTS

We would like to thank Christoph Herrmann for his comments on an earlier draft, and the anonymous reviewers for their helpful suggestions. This work is generously supported by EPSRC grant EP/C001346/1, and by EU Framework VI Grant IST-2004-510255, funded under the FET-Open programme.

REFERENCES

[1] P. Amey. Correctness by Construction: Better can also be Cheaper. *CrossTalk: the Journal of Defense Software Engineering*, pages 24–28, March 2002.

[2] E. Axelsson and K. Claessen M. Sheeran. Wired: Wire-aware circuit design. In *CHARME 2005*, 2005.

[3] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in Haskell. In *Proc. ICFP '98*, 1998.

[4] Edwin Brady. *Practical Implementation of a Dependently Typed Functional Programming Language*. PhD thesis, University of Durham, 2005.

[5] Edwin Brady. Ivor, a proof engine. In *Proc. Implementation of Functional Languages (IFL 2006)*, volume 4449 of *LNCS*. Springer, 2007. To appear.

[6] Edwin Brady and Kevin Hammond. A Verified Staged Interpreter is a Verified Compiler. In *Proc. ACM Conf. on Generative Prog. and Component Engineering (GPCE '06), Portland, Oregon*, 2006.

[7] Edwin Brady, Conor McBride, and James McKinna. Inductive families need not store their indices. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs 2003*, volume 3085, pages 115–129. Springer, 2004.

[8] James Chapman, Thorsten Altenkirch, and Conor McBride. Epigram reloaded: a standalone typechecker for ETT. In *TFP*, 2005.

[9] Coq Development Team. The Coq proof assistant — reference manual. `http://coq.inria.fr/`, 2001.

[10] J. Eckhardt, R. Kaibachev, E. Pašalíc, K. Swadi, and W. Taha. Implicitly Heterogeneous Multi-Stage Programming. In *Proc. 2005 Conf. on Generative Programming and Component Engineering (GPCE 2005)*, Springer-Verlag LNCS 3676, 2005.

[11] J. Grundy, T. Melham, and J. O'Leary. A Reflective Functional Language for Hardware Design and Theorem Proving. *J. Functional Programming*, 16(2):157–196, 2006.

[12] K. Hammond, G. Grov, G. J. Michaelson, and A. Ireland. Low-Level Programming in Hume: an Exploration of the HW-Hume Level. Submitted to IFL '06, 2006.

[13] Christoph A. Herrmann. Generating message-passing programs from abstract specifications by partial evaluation. *Parallel Processing Letters*, 15(3):305–320, 2005.

[14] Christoph A. Herrmann. Type-sensitive size parametrization of circuit designs by metaprogramming. Technical Report MIP-0601, Universität Passau, February 2006.

[15] John Hughes. Why functional programming matters. Technical Report 16, Programming Methodology Group, Chalmers University of Technology, November 1984.

[16] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programmin*, pages 410–423, 1996.

[17] Xavier Leroy. Formal certification of a compiler back-end. In *Principles of Programming Languages 2006*, pages 42–54. ACM Press, 2006.

[18] Pierre Letouzey. A new extraction for Coq. In Herman Geuvers and Freek Wiedijk, editors, *Types for proofs and programs*, LNCS. Springer, 2002.

[19] Zhaohui Luo. *Computation and Reasoning – A Type Theory for Computer Science*. International Series of Monographs on Computer Science. OUP, 1994.

[20] Conor McBride. *Dependently Typed Functional Programs and their proofs*. PhD thesis, University of Edinburgh, May 2000.

[21] Conor McBride. Epigram: Practical programming with dependent types. Lecture Notes, International Summer School on Advanced Functional Programming, 2004.

[22] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A proof assistant for higher order logic*, volume 2283 of *LNCS*. Springer-Verlag, March 2002.

[23] Christine Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions*. PhD thesis, Paris 7, 1989.

[24] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple Unification-Based Type Inference for GADTs. In *Proc. ICFP '06: 2006 International Conf. on Functional Programmin g*, 2006.

[25] William Pugh. The Omega Test: a fast and practical integer programming algorithm for dependence analysis. *Communication of the ACM*, pages 102–114, 1992.

[26] W. Taha. A Gentle Introduction to Multi-stage Programming, 2003. Available from http://www.cs.rice.edu/ taha/publications/journal/dspg04a.pdf.

[27] P. Wadler. Theorems for free! In *Proc. 4th Int. Conf. on Funct. Prog. Languages and Computer Arch., FPCA'89, London, UK, 11–13 Sept 1989*, pages 347–359. ACM Press, New York, 1989.

[28] Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University, December 1998.