M. Methodology

M.1 Implementing Addition

The PVS source file k.pvs illustrates basic concepts of implementation verification using binary addition as an example.

M.1.1 Review of Binary Addition

You might never have learned, or may not recall, how binary addition works. Section 2.5 of Induction, Recursion and Programming describes this in detail. More briefly, binary addition is done in the same way as decimal addition, column by column. The only difference is that the base is 2 rather than 10. Each column is summed and if the column-sum exceeds a single digit the leading 1 is carried to the next column.

\[
\begin{array}{cccccc}
1 & 1 & 1 & 0 & 0 & 1 \\
\downarrow & + & \downarrow & \downarrow & \downarrow & \downarrow \\
0 & 1 & 0 & 1 & 0 & 1 \\
\downarrow & + & \downarrow & \downarrow & \downarrow & \downarrow \\
1 & 0 & 0 & 0 & 1 & 1 \\
\end{array}
\]

Implementing Binary Addition The goal is to describe implementation of this algorithm in boolean logic, using operators ‘·’ for logical and, ‘+’ for logical or, and \(\overline{x}\) for not. Such an implementation would have identical components for each column.

The carry bit \(c_{i+1}\) is 1 whenever two or more of the inputs are 1s, that is,

\[
c_{i+1} = \text{majority}(a_i, b_i, c_i) \overset{\text{def}}{=} a_i \cdot b_i + a_i \cdot c_i + b_i \cdot c_i
\]

The sum bit \(s_i\) is 1 when an odd number of inputs are 1s, that is,

\[
s_i = \text{parity}(a_i, b_i, c_i) \overset{\text{def}}{=} a_i \oplus b_i \oplus c_i
\]

where ‘⊕’ stands for exclusive-or,

\[
x \oplus y \overset{\text{def}}{=} x \cdot \overline{y} + \overline{x} \cdot y
\]
M.1.2 Implementation Verification

Recall from the Terminology notes, that verification is described as the process of determining whether an implementation satisfies specification. In this example, the specification is, “add two natural numbers,” and the implementation is to perform binary addition on two strings of binary digits, or numerals. So the key detail added in this implementation is the representation of numbers by binary numerals.

\[
\begin{align*}
\text{number}^2 & \xrightarrow{+} \text{number} \\
\rho & \xrightarrow{\alpha} \rho & \xrightarrow{\alpha} \\
\text{numeral}^2 & \xrightarrow{\text{adder}} \text{numeral}
\end{align*}
\]

The functions \( \rho \) and \( \alpha \) relate numbers and numerals. Given an number \( n \) and a numeral \( N = d_k \cdots d_1 d_0 \), the abstraction function \( \alpha \) in the diagram is defined

\[
\alpha[d_0 d_1 \cdots d_k] = \sum_{i=0}^{k} 2^i \tilde{d}_i
\]

On the right-hand side, digit \( d_i \) has been decorated \( \tilde{d}_i \) because it is being interpreted as a number rather than a symbol: \( \tilde{0} \leftrightarrow 0 \) and \( \tilde{1} \leftrightarrow 1 \). We do not need to define a representation function (\( \rho \), see Note 1) because the form of our correctness statement is

For all \( X, Y \in \text{numeral} \), \( \alpha[\text{adder}(X, Y)] = \alpha[X] + \alpha[Y] \)

In words, “All representable numbers are added correctly.”

\[
\begin{align*}
\alpha(X), \alpha(Y) & \xrightarrow{\text{adder}} \alpha(X) + \alpha(Y) \\
X, \ Y & \xrightarrow{\text{adder}(X, Y)}
\end{align*}
\]

M.1.3 Formulation in PVS

Numerals are modeled as inductively defined boolean lists.

```pvs
boolist: DATATYPE
BEGIN
null: null?
cons (first: bool, rest:boolist):cons?
END boolist
```

The primitive `bool` type is used to model binary digits (bits) so that PVS’s logical operations `AND`, `OR`, `NOT`, `XOR`, etc. may be used to formulate the majority and parity functions defined earlier. A `boolist` represents a binary numeral.
whose leading digit is the least significant bit. For example, the binary numeral
1011 is expressed as

\[
\text{cons(true, cons(true, cons(false, cons(true, null)))})
\]

The abstraction function \( \alpha \) can then be easily defined recursively as

\[
\text{VAL}(l:\text{boolist}): \text{RECURSIVE} \text{ nat} =
\begin{cases}
\text{null}: 0, \\
\text{cons}(b, tl): (\text{IF } b \text{ THEN } 1 \text{ ELSE } 0 \text{ ENDIF}) + 2 * \text{VAL}(tl)
\end{cases}
\]

MEASURE \( l \) by <<

M.1.4 Notes

1. Let ‘÷’ stand for integer quotient. The representation function \( \rho \) would be

\[
\rho(n) = d_k \cdots d_1 d_0 \text{ where } k \geq (\log_2 n) \text{ and } d_i = \begin{cases}
0 & \text{if } (n \div 2^i) \text{ is even} \\
1 & \text{if } (n \div 2^i) \text{ is odd}
\end{cases}
\]

2. The INC example recursively traverses a boolist.

\[
\text{INC}(l:\text{boolist}): \text{RECURSIVE} \text{ boolist} =
\begin{cases}
\text{null}: \text{cons(true, null)}, \\
\text{cons}(b, tl): \text{IF } b \text{ THEN \text{cons(false, INC}(tl)) ELSE cons(true, tl) ENDIF}
\end{cases}
\]

MEASURE \( l \) by <<

Whether this suggest temporal (“bit-serial”) or geometric (“bit-parallel”) iteration open to interpretation. It depends on what the recursion is intended to model.

3. The suggested ADD function,

\[
\text{ADD}(l1, l2: \text{boolist}, c: \text{bool}): \text{RECURSIVE} \text{ boolist} = \ldots
\]

does not require the boolist arguments to be the same length. This is a bit simpler to deal with, but one would ordinarily expect to see an \( N \)-bit adder, for some fixed constant \( N \).