

Functional Pearl: Sparse Bitmaps for Pattern Match Coverage *

Ki Yung Ahn Tim Sheard

Portland State University

kya@cs.pdx.edu sheard@cs.pdx.edu

Abstract

Pattern matching coverage over Algebraic Data Types (ADTs) has most often been studied in the context of pattern compilation algorithms. However, it is worth considering the pattern matching coverage problem in isolation, since general solutions will be independent of the specifics of any implementation or language.

We define an intuitive and mathematically well-established bit masking semantics for pattern match coverage. We design and implement a sparse bitmap data structure, which realizes this semantics in a compact and flexible manner. This bitmap data structure supports computing coverage solutions of large programs incrementally from coverage solutions of sub-programs. It can also be used as a common data representation for pattern coverage shared between different tools (e.g., compilers, linting tools, software analysis tools) that need pattern match coverage information.

Categories and Subject Descriptors D.3.3 [Programming Languages]: Language Constructs and Features—Patterns; E.2 [Data]: Data Storage Representations—Linked Representations; F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs—Functional Constructs

General Terms Languages, Theory, Algorithms, Verification

Keywords datatypes, algebraic data types, pattern matching coverage, exhaustive patterns, bitmask, bitmap, boolean tensor, tensor product, type directed computation, evaluation strategy, type invariants, type preserving computation

1. Introduction

Functional language implementations have traditionally treated pattern match coverage checking as a by-product of pattern compilation algorithms. Typical approaches to pattern compilation use decision trees (Cardelli 1984; Pettersson 1992) or backtracking (Augustsson 1985). However, approaches to the pattern match coverage problem in isolation have also generated interest (Maranget 2007). The benefit of this independent approach is that we can apply general solutions independent of the specifics of particular implementations or languages. Recent experimental languages and experimental features in functional languages demonstrate why this kind of study is important. They often require more *general* and

composable solutions for the pattern matching coverage problem than the approaches enabled by the pattern compilation process. For example, the pattern matching semantics of functional logic languages like Curry (Hanus 2006) are not compatible with sequential pattern compilation algorithms. Consider the question whether the Curry function f is total, where f is defined as $f\ x = f1\ x\ ?\ f2\ x$ using the non-deterministic choice operator $?$. Here, the result of f can be successfully defined over all input x when the coverage of $f1$ union the coverage of $f2$ is exhaustive, even if the coverage of neither $f1$ nor $f2$ is exhaustive by itself. In order to give good answers to such questions, we need a pattern match coverage semantics that is not only *general*, in the sense that it should be independent of specific pattern compilation algorithms, but also *composable*, in the sense that it should be able to reason about exhaustiveness of large programs by composing exhaustiveness results from sub-programs incrementally.

Our contributions are the following:

1. We give each pattern an intuitive coverage semantics using bitmap-like structures that are composable and mathematically well established (§2).
2. We design a sparse bitmap data structure that implements the coverage semantics in a compact representation yet which is able to expand its size and accuracy dynamically as needed (§3). Using our sparse bitmaps, we can
 - (a) Decide whether the patterns in a single case expression are exhaustive for matching all values whose types are defined using a polymorphic ADTs. Note, this is independent of a specific pattern compilation strategy.
 - (b) Provide a composable data representation for handling partial coverage of patterns for monomorphic ADTs, which we may use it as a concrete basis for representing coverage of larger parts of programs (e.g., functions calling several other functions with partial coverage).
3. We provide a concise reference implementation of the sparse bitmap data structure and its operations in Haskell, whose full implementation fits in less than one page (§3). This is one component of the “pearl”.
4. We study several interesting aspects regarding the representation of sparse bitmap operations while trying to re-implement a type safe and polymorphic version of the sparse bitmap data structure (§4). We use advanced Haskell features to encode type preserving computation and type directed computation. We also identify some limitations of current Haskell implementations (e.g., type invariants; interaction among GADT, polymorphism and type classes). This is the second component of the “pearl”.

* Supported by the NSF Grants No. CCF-0541447 and 0613969

2. A coverage semantics using bitmaps

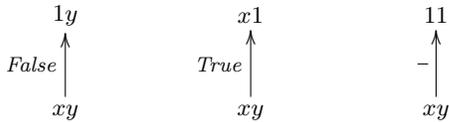
We use bitmaps to check whether the coverage of patterns is exhaustive. We start from a clean bitmap of all zero bits, and mark the bits properly as we observe the patterns. If we end up with a bitmap of all bits set to one, we conclude that the pattern match coverage was exhaustive. The number of bits we need depends on the number of constructors in the type, the types of the arguments to data constructors, and the nesting depth of the patterns. We consider these issues one at a time.

2.1 Types consisting only of nullary data constructors

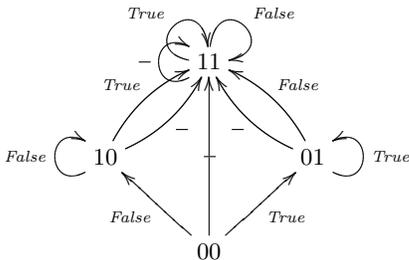
For a data type consisting of only nullary data constructors (data constants), we need exactly as many bits as the number of nullary constructors in that type. That is, we need n bits for the type consisting of n data constants. For example, we need two bits for the boolean value type *Bool* with the two data constructors *False* and *True*, which we can define using the Haskell (Peyton Jones et al. 2003) notation as follows:

```
data Bool = False | True
```

There are three possible patterns with the type *Bool*: *False*, *True*, and $_$ (the wildcard pattern).¹ We think of patterns as bitstring transformers, mapping bitstrings to bitstrings. *False* maps the first bit to 1, *True* maps the second bit to 1, and $_$ maps both bits to 1. We may visualize this as follows:



It is easy to see that any path from 00 to 11 in the following diagram is a proof of the fact that a list of the patterns along that path form an exhaustive cover:



For example, $00 \Rightarrow 11$ is a proof that the list $\langle _ \rangle$ is an exhaustive cover, and $00 \xrightarrow{False} 10 \xrightarrow{True} 11$ is a proof that the list $\langle False, True \rangle$ forms an exhaustive cover. Because we (1) interpret patterns as bitstring transformers; and (2) these transformers always transform a fixed set of bits to one, leaving the other bits untransformed, and (3) the transformers never switch positions among the bits, we may write our proofs more compactly as compositions: $True \circ False$. Note also, that the three conditions listed above, imply that each proof step is commutative. For example, $False \circ True = True \circ False$.

Exploiting these convenient invariants, we can represent the bitstring transformers induced by patterns as *bitmaps* (or *bitmasks*), which are plain sequences of bits, which interpret 1's to mean flip this bit to 1, and 0's to mean leave this bit unchanged. For example, we represent *False* as 10, *True* as 01, and $_$ as 11. We can also interpret composition (\circ) as the bitwise-or operation. This suggests that an appropriate algorithm for testing the completeness coverage

¹ We need not consider variable patterns since they are equivalent to $_$ in terms of coverage.

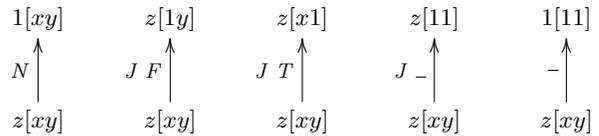
of a list of patterns is to bitwise-or the bitstrings for each pattern, and then test to see if the final result is a bitstring composed of all 1s.

2.2 Coverage over types with unary data constructors

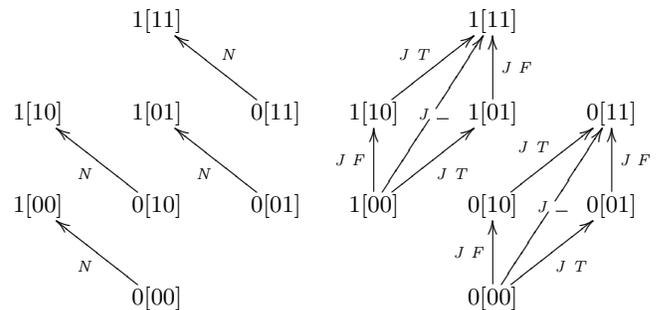
The previous example on *Bool* suggests we need at least n bits for pattern match coverage checking over the patterns of an ADT with n data constructors since each bit corresponds to the usage of each data constructor of the ADT. When data constructors have arguments, we need additional bits to check the coverage over those argument patterns. The mechanism we use to represent these additional bits is key to the approach we use; it is best introduced by example. Consider the *Maybe* type in Haskell:

```
data Maybe a = Nothing | Just a
```

Since *Just* takes an argument, it is not enough to simply check whether we have used both constructors of *Maybe* to decide exhaustiveness. For example, *Nothing* and $(Just\ True)$ do not form an exhaustive cover, while *Nothing* and $(Just\ _)$ do. In this section we continue to consider the coverage over patterns of type (*Maybe Bool*). For succinctness, we abbreviate the patterns *Nothing*, *Just*, *True*, and *False* as *N*, *J*, *T*, and *F*. Because the constructor *Just* takes a single argument, the single bit representing *Just* is generalized itself to a bitstring² The length of the generalized bit for *Just* depends only upon the type of the argument to *Just*. If the argument has type *Bool* then the bit for *Just* is generalized to a bitstring of length 2 (because *Bool* has two nullary constructors). Thus each pattern of type (*Maybe Bool*) manipulates the (generalized) bitstring as follows:



Note, *J* is a homomorphism from the patterns of *Bool* to the patterns of *Maybe Bool*. For example, $J\ F \circ J\ T = J\ (F \circ T)$. Since the state space of *Maybe Bool* is twice as large as that of *Bool*, it gets very messy to draw a complete transition diagram as we did for *Bool*. So, we illustrate *Nothing* and *Just* patterns in two separate diagrams as follows:³



² A (generalized) bitstring is no more than a tree of bits, as we shall see its Haskell implementation in §3. However, referring them as bitstrings rather than bit-trees makes it more natural to explain the concept of the tensor product (\otimes), which we shall shortly see in §2.4, since tensor products are usually defined on vectors or matrices rather than trees.

³ You can overlap the two diagrams in your mind and add the paths for the wildcard ($_$) (from any vertex to $1[11]$) and repetitions (self cycle) to get the complete diagram for *Maybe Bool*.

	-	Z	S -	S Z	S (S -)
0	1				
1	11	10	01		
2	1[11]	1[00]	0[11]	0[10]	0[01]
3	1[1[11]]	1[0[00]]	0[1[11]]	0[1[00]]	0[0[11]]

Table 1. Coverage of *Nat* patterns depending on max nesting depth. Note as the depth increases, bits within generalized bits themselves become generalized.

2.3 Coverage over recursive types

Patterns of recursive types also need additional bits. Here the number of additional bits depends upon the maximum nesting depth of the pattern in the inductive arguments. Although the cause is arguably different, the result is the same; additional bits are introduced by expanding single bits into bitstrings. The inductive nature of a recursive datatype allows the bitstring in the expanded bit to contain expanded bits. The depth of this expansion corresponds exactly to the nesting depth of the pattern. For example, consider the unary natural number type *Nat*:

```
data Nat = Zero | Succ Nat
```

We abbreviate the patterns *Zero* and *Succ* as *Z* and *S* for succinctness. We say, for example, *Z* has depth 1, *S Z* has depth 2, and *S (S Z)* has depth 3. We consider the wildcard pattern to have depth 0. For example *-* has depth 0, *S -* has depth 1, and *S (S -)* has depth 2. Table 1 shows how we can represent the coverage of natural number patterns when the maximum nesting depth is given. For example, the representation of *Z* is 10, 1[00], or 1[0[00]] when the maximum nesting depth is 1, 2, or 3, respectively.

2.4 Coverage over types with *n*-ary data constructors

When dealing with types that have data constructors taking multiple arguments we also need additional bits beyond those required for the *n* constructors. Again, individual bits are generalized, but the nature of the expansion becomes more complex. The bit corresponding to a constructor with two arguments is generalized to a 2 dimensional matrix. The bit corresponding to a constructor with three arguments is generalized to a 3 dimensional matrix, etc. The size of these multidimensional matrices depends only on the type of each argument. This complexity arises because we need more care to check exhaustiveness of the patterns of such types.

For example, the simplest two argument patterns are the patterns of the type $(Bool, Bool)$, constructed with the pairing pattern constructor $(,)$ which takes two boolean sub-patterns. When is a list of such patterns exhaustive? The list $\langle (F, T), (T, F) \rangle$ is not exhaustive, but the list $\langle (-, F), (-, T) \rangle$ is exhaustive. How can we explain this subtlety using generalized bitstrings?

The pairing type has a single constructor $(,)$, so its bitstring representation will be a bitstring of length 1. It takes two arguments, so the single bit in that bitstring will be a 2-dimensional matrix. At the type $(Bool, Bool)$, the matrix will be a 2×2 matrix, since the bitstring representing *Bool* patterns has 2 bits. We list the possible bitstring representations below.

$$\frac{p \mid (F, F) \quad (F, T) \quad (T, F) \quad (T, T) \quad -}{\llbracket p \rrbracket \mid \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \quad \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}}$$

$$\frac{p \mid (-, F) \quad (-, T) \quad (F, -) \quad (T, -) \quad (-, -)}{\llbracket p \rrbracket \mid \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix} \quad \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}}$$

We want to emphasize that each matrix is a bitstring of length 1, where that single bit has been generalized into a 2×2 matrix. Why are these the correct matrices? At the end of §2.1 we suggested that the appropriate algorithm for testing the completeness of two

patterns was to bitwise-or the bitstrings for each pattern, and test the result to see if it is a bitstring composed of all 1s. We use this intuition to explain why the two dimensional expanded bits above are correct.

A value of type $(Bool, Bool)$ can have one of four possible shapes. Each shape corresponds to a bit of 1 in one of the four quadrants of the 2×2 matrix: $(False, False)$ upper left, $(False, True)$ upper right, $(True, False)$ lower left, and $(True, True)$ lower right. One can read these off from the table above. So, a list of patterns has complete coverage only if all four quadrants are set to 1. Patterns with a wildcard will have several quadrants set to 1, as these will match more than one of the four possible values. Now, it is easy to see why the list of patterns $\langle (-, F), (-, T) \rangle$ is exhaustive, but the list of patterns $\langle (F, T), (T, F) \rangle$ is not exhaustive. Try it yourself; bitwise-or the appropriate matrices and see.

Now that we are convinced these are the correct matrices, how do we compute $\llbracket (p_1, p_2) \rrbracket$ from $\llbracket p_1 \rrbracket$ and $\llbracket p_2 \rrbracket$ where $\llbracket p \rrbracket$ denotes the bitmap representation for the coverage of *p*?

We calculate $\llbracket (p_1, p_2) \rrbracket$ in terms of $\llbracket p_1 \rrbracket$ and $\llbracket p_2 \rrbracket$ using the well known mathematical concept called *tensor product* (or *Kronecker product*) (Lang 2002). The formula for the tensor product of two bit vectors of size 2 is:

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \otimes \begin{pmatrix} y_1 & y_2 \end{pmatrix} = \begin{pmatrix} x_1 \&y_1 & x_1 \&y_2 \\ x_2 \&y_1 & x_2 \&y_2 \end{pmatrix} \quad (1)$$

Thus the correct formula for patterns of type $(Bool, Bool)$ is

$$\llbracket (p_1, p_2) \rrbracket = \llbracket p_1 \rrbracket \otimes \llbracket p_2 \rrbracket$$

For example,

$$\llbracket (T, F) \rrbracket = \llbracket T \rrbracket \otimes \llbracket F \rrbracket = 01 \otimes 10 = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$$

In this way, we can derive other patterns of $(Bool, Bool)$ that contain wildcard patterns:

$$\llbracket (-, F) \rrbracket = \llbracket - \rrbracket \otimes \llbracket F \rrbracket = 11 \otimes 10 = \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix}$$

Interpreting \circ as bitwise-or operation $|$ as we did for *Bool*, we can see that the intuitive properties, such as the coverage of $(-, F)$ is same as the union of the coverages of (F, F) and (T, F) , hold:

$$\begin{aligned} \llbracket (F, F) \circ (T, F) \rrbracket &= \llbracket (F, F) \rrbracket \mid \llbracket (T, F) \rrbracket \\ &= \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \mid \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} = \llbracket (-, F) \rrbracket \end{aligned}$$

Similarly, we can encode any boolean triple pattern of type $(Bool, Bool, Bool)$ with 8 bits, or $2 \times 2 \times 2$ three dimensional bitmap. For example,

$$\llbracket (True, -, False) \rrbracket = \llbracket True \rrbracket \otimes \llbracket - \rrbracket \otimes \llbracket False \rrbracket = 01 \otimes 11 \otimes 10$$

Of course, picturing a three dimensional matrix on a flat display is somewhat problematic, so we resort to the following tricks. We assume that tensor products associate to the right. That is, we consider $(n + 1)$ -tuples to be isomorphic to the pairs of the first component and the *n*-tuple consisting of the other components (i.e., $\llbracket (x_0, x_1, \dots, x_n) \rrbracket = \llbracket (x_0, (x_1, \dots, x_n)) \rrbracket$). And, we represent multi-dimensional matrices as lists of lists of lists etc.

$$01 \otimes 11 \otimes 10 = 01 \otimes [10][10] = \llbracket [00][00] \rrbracket \llbracket [10][10] \rrbracket$$

More generally, the coverage for an *n*-tuple pattern is a tensor product of the coverages for its components:

$$\llbracket (p_0, \dots, p_n) \rrbracket_{(t_0, \dots, t_n)} = \llbracket p_0 \rrbracket_{t_0} \otimes \dots \otimes \llbracket p_n \rrbracket_{t_n}$$

In general, when we have an ADT of the following definition,

```
data T = C0 s0 ... sn0 | ... | Ck t0 ... tnk | ... | Cm u0 ... unm
```

the coverage for the pattern $C_k p_0 \dots p_{n_k}$ of type T is:

$$\llbracket C_k p_0 \dots p_{n_k} \rrbracket_T = \mathbf{0}_{(s_0, \dots, s_{n_0})} \cdots \llbracket (p_0, \dots, p_{n_k}) \rrbracket_{(t_0, \dots, t_{n_k})} \cdots \mathbf{0}_{(u_0, \dots, u_{n_m})}$$

where $\mathbf{0}_t$ is the sequence of all zero bits whose size is equal to the coverage of the patterns of type t . And, the coverage for the wildcard pattern $_$ of type T is as follows:

$$\llbracket _ \rrbracket_T = \mathbf{1}_T = \mathbf{1}_{(s_0, \dots, s_{n_0})} \cdots \mathbf{1}_{(t_0, \dots, t_{n_k})} \cdots \mathbf{1}_{(u_0, \dots, u_{n_m})}$$

where $\mathbf{1}_t$ is the sequence of all one bits whose size is equal to the coverage of the patterns of type t .

2.5 Extended example: coverage over *Maybe* (*Nat*, [*Bool*])

Let us consider a comprehensive example which incorporates everything we discussed in this section. In Figure 1, we display the bitstring representation of six patterns of type *Maybe* (*Nat*, [*Bool*]). We can see that these six patterns form an exhaustive cover, because at least one of them has the bit set for every position. That is, we can decide the exhaustiveness of the patterns by performing bitwise-or operation over the coverage of all the patterns.

We can also observe that the second pattern $J (_, [])$ and the third pattern $J (Z, _)$ are overlapping but all the other patterns are non-overlapping. Of all the patterns only $J (_, [])$ and $J (Z, _)$ have a common 1 in the same position. That is, we can decide whether two patterns are overlapping by performing bitwise-and operation over their coverage, and testing if the result contains a 1.

Note, the bitmaps in Figure 1 are much more complex than the bitmaps in Table 1. The nesting structure of bitmaps gets quite complicated even for relatively simple instances of parameterized types when they involve types with constructors taking multiple arguments, or the patterns themselves become deeply nested.

In fact, the complexity of the bitmap for a particular pattern can vary depending upon the other patterns we want to compare it with. For example, in Figure 1, we cannot easily recognize that the proper representation for the first pattern $\llbracket N \rrbracket$ is $1[[00][0[[[00][00]]][[[00][00]][[00][00]]]]]$, instead of 10 or $1[00]$, until we have calculated the coverage representation for the fifth pattern as follows:

$$\begin{aligned} \llbracket J (S _, T : _) \rrbracket &= 0(\llbracket S _ \rrbracket \otimes \llbracket T : _ \rrbracket) \\ &= 0(0[_] \otimes 0(\llbracket T \rrbracket \otimes [_])) \\ &= 0(0[_] \otimes 0(01 \otimes 11)) \\ &= 0(0[_] \otimes 0(01 \otimes 11)) \\ &= 0[0[1[11]] \otimes 0[[00][11]]] \\ &= 0[[00][0[1[11]] \otimes [00][11]]] \\ &= 0[[00][0[[[00][11]][[11 \otimes 00][11 \otimes 11]]]] \\ &= 0[[00][0[[[00][11]][[[00][00]][[11][11]]]]] \end{aligned}$$

Thus we need a representation for bitmaps that can dynamically grow and shrink as we compute with it. In the following section, we discuss the implementation of a bitmap data structure that can start from a minimal representation (e.g., $\llbracket N \rrbracket = 10$) and expand to more complex representations (e.g., $\llbracket N \rrbracket = 1[00]$ or $\llbracket N \rrbracket = 1[[00][00]]$) as needed.

3. Sparse bitmaps of variable size and depth

The bitmap structure for representing the coverage of patterns has the following two characteristics: (1) bitmap structures consisting of only 0s or only 1s appear quite often, and (2) one pattern can have many representations of different size and depth.

The first characteristic has two causes: the tensor product and the wildcard pattern. Tensor products produce bit blocks of consecutive 0s. For example, the tensor product $100 \otimes 010 =$

$$\begin{aligned} \llbracket N \rrbracket &= 1[[00][0[[[00][00]][[[00][00]][[00][00]]]]] \\ \llbracket J (_, []) \rrbracket &= 0[[10][1[[[00][00]][[[00][00]][[00][00]]]]] \\ \llbracket J (Z, _) \rrbracket &= 0[[11][0[[[00][00]][[[00][00]][[00][00]]]]] \\ \llbracket J (S Z, F : _) \rrbracket &= 0[[00][0[[[11][00]][[[00][00]][[00][00]]]]] \\ \llbracket J (S _, T : _) \rrbracket &= 0[[00][0[[[00][11]][[[00][00]][[11][11]]]]] \\ \llbracket J (S (S _), F : _) \rrbracket &= 0[[00][0[[[00][00]][[[11][11]][[00][00]]]]] \end{aligned}$$

Figure 1. Coverage for the patterns of type *Maybe* (*Nat*, [*Bool*])

$[010][000][000]$ produces two blocks of 000. The wildcard pattern produces coverage representations of consecutive 1s by definition.

The second characteristic also has two causes: the wildcard pattern and constructors of recursive/parameterized types. Representations for the wildcard pattern depend on the type that the wildcard pattern is used at. For example, $\llbracket _ \rrbracket_{Bool} = 11$ and $\llbracket _ \rrbracket_{Maybe Bool} = 1[11]$. Similarly, patterns of parameterized types (e.g., *Nothing* :: *Maybe a*) can have different representations depending on how its type parameters are instantiated. For example, $\llbracket N \rrbracket_{Maybe Bool} = 1[00]$ while $\llbracket N \rrbracket_{Maybe (Maybe Bool)} = 1[0[00]]$. Patterns of recursive types (e.g., *Zero* :: *Nat*) may also have varying representations depending on the maximum nesting depth, as we have already seen in Table 1. These varying representations differ only in their size and depth, but share the common structure of the pattern which they originate from.

We design and implement a bitmap data structure that (1) handles consecutive 1s and 0s in a more compact manner analogous to sparse matrix implementations and (2) focuses on the fundamental structure of generalized bitstrings while being flexible enough to expand its size and depth as needed. In the following subsection, we explain the details of the design and implementation of the sparse bitmap data structure for pattern match coverage.

3.1 The sparse bitmap data structure *BIT*

Figure 2 shows the full Haskell implementation of the sparse bitmap data structure *BIT* and its operations, except the definitions related to the value representation of type information. The type representation *Type* and its operations are shown separately in Figure 3.

3.1.1 The *BIT* data type

$0 t$ stands for bitmaps of all 0s with any appropriate size and depth for the coverage of type t . For example, $0 (Bool, Bool)$ can mean 0, 00, or $[00][00]$. In other words, $0 t$ represents the null coverage for type t . Note, $0 t$ is an identity for the bitwise-or operation (\cdot).

Similarly, $1 t$ stands for bitmaps all 1s with any appropriate size and depth for the coverage of type t . For example, $1 (Bool, Bool)$ can mean 1, 11, or $[11][11]$. In other words, $1 t$ represents the full coverage for type t . Note, $1 t$ is an identity for the bitwise-and operation ($\cdot \&$).

The data constructor *Bs* allows us to consider a list of *BIT*s as a *BIT*. The need for *Bs* arises from the constructors that have arguments. For example, the coverage of *Just False* is $0[10]$, which we can describe using the *BIT* data type as follows:

$$Bs [0 \textit{unit}, Bs [1 \textit{unit}, 0 \textit{unit}]] Bool \textit{ (Maybe Bool)}$$

Here, we can see that $0 \textit{unit}$ and $Bs [1 \textit{unit}, 0 \textit{unit}] Bool$ lie at the same level. *Type* values such as *Bool* and *unit* are defined in Figure 3. *Bool* :: *Type* is the value representation of the Haskell type *Bool*, as its name suggests. *unit* stands for data constants which we cannot expand to any deeper structure. We will discuss *Type* values in more depth later on, when we explain some of the other definitions that use *Type* values.

```

data BIT
  = 0 Type          -- all 0 bits (identity on (.|))
  | I Type          -- all 1 bits (identity on (.&))
  | Bs [BIT] Type  -- row of bits (possibly nested)

typeof      :: BIT → Type
typeof (0 t) = t
typeof (I t) = t
typeof (Bs _ t) = t

(.|)          :: BIT → BIT → BIT
0 t .| y      = y
x .| 0 t      = x
I t .| _      = I t
_ .| I t      = I t
Bs xs t .| Bs ys _ = Bs (xs .|. ys) t
(.|.) :: [BIT] → [BIT] → [BIT]
(.|.) = zipWith (.|)

(.&)          :: BIT → BIT → BIT
0 t .& _      = 0 t
_ .& 0 t      = 0 t
I t .& y      = y
x .& I t      = x
Bs xs t .& Bs ys _ = Bs (xs .&. ys) t
(.&.) :: [BIT] → [BIT] → [BIT]
(.&.) = zipWith (.&)

infixr 7 ⊗
(⊗)          :: BIT → BIT → BIT
x ⊗ 0 u      = 0 (t ⊗ u) where t = typeof x
0 t ⊗ y      = 0 (t ⊗ u) where u = typeof y
I t ⊗ I u    = I (t ⊗ u)
Bs xs t ⊗ I u = case expand u of
  [] → Bs xs t
  us → Bs xs t ⊗ Bs (map I us) u
I t ⊗ Bs ys u = case expand t of
  [] → Bs ys u
  ts → Bs (map I ts) t ⊗ Bs ys u
Bs xs t ⊗ Bs ys u = Bs (xs ⊗ ys) (t ⊗ u)
where
  xs ⊗ ys = [Bs [x ⊗ y | y ← ys] (typeof x ⊗ u) | x ← xs]

reduce       :: BIT → BIT
reduce (Bs xs t) = case map reduce xs of
  0 _ : xs | all o xs → 0 t
  I _ : xs | all i xs → I t
  xs'      → Bs xs' t
where o (0 _) = True
      o _     = False
      i (I _) = True
      i _     = False

reduce x     = x

```

Figure 2. The *BIT* data structure and its operations: bitwise-or ($.|$), bitwise-and ($.&$), tensor-product (\otimes), and *reduce*.

```

data Type = Bool | Color | Nat
           | Maybe Type | List Type
           | Tuple [Type]
-- Tuple [t] is junk and should never occur.
-- Tuple [] is also junk, but we use it to indicate
-- a nullary constructor. See expand below.

data Nat = Zero | Succ Nat
data Color = Red | Green | Blue

unit = Tuple []

expand      :: Type → [Type]
expand Bool = [unit, unit]
expand Color = [unit, unit, unit]
expand Nat  = [unit, Nat]
expand (Maybe a) = [unit, a]
expand (List a) = [unit, a ⊗ List a]
expand (Tuple []) = []
expand (Tuple (x : xs)) =
  case expand x of
    [] → expand (Tuple xs)
    ts → [foldr (⊗) unit (t : xs) | t ← ts]

infixr 7 ⊗
(⊗) :: Type → Type → Type
t ⊗ u | null (expand t) = u
      | null (expand u) = t
      | otherwise       = Tuple [t, u]

```

Figure 3. The value representation of types and its operations: *expand*, and *Type* product (\otimes)

3.1.2 Type information of BITs

Each *BIT* contains both *Type* and bitstring information. For example, the representation of $\llbracket \text{False} \rrbracket$ is $(Bs [I \text{unit}, 0 \text{unit}] \text{Bool})$ where $[I \text{unit}, 0 \text{unit}]$ is the bitstring information and *Bool* is the *Type* information. We can extract *Type* values from *BIT*s using the function $typeof :: BIT \rightarrow Type$ defined in Figure 2. For example, $typeof (0 \text{Bool}) = \text{Bool}$, $typeof (I \text{Bool}) = \text{Bool}$, and $typeof (Bs [I \text{unit}, 0 \text{unit}] \text{Bool}) = \text{Bool}$. We need this *Type* information to define tensor product (\otimes) correctly, as we shall see in §3.1.4. We may use this type information to compute missing patterns (patterns that instantiate to unmatched values) from a non-exhaustive list of patterns. We may also use the type information to check whether a *BIT* value is *well-formed* with respect to its type information. For example, $Bs [I \text{unit}, 0 \text{unit}] \text{Bool}$ is well-formed since *Bool* consists of two data constants, but $Bs [I \text{unit}, 0 \text{Nat}] \text{Bool}$ and $Bs [I \text{unit}, 0 \text{unit}, I \text{unit}] \text{Bool}$ are not. In general, for any given *Type* value t ,

1. $0 t$ and $I t$ are always well-formed,
2. $Bs [] t$ is never well-formed, and
3. $Bs [b_1, \dots, b_n] t$ is well-formed when each b_i is well-formed and $expand t = map\ typeof [b_1, \dots, b_n]$.

The function *expand* in Figure 3 encodes information about the shape of each *Type* from its declaration. For example, the third equation $expand \text{Bool} = [unit, unit]$ means that *Bool* was declared to have two nullary data constructors *False* and *True*, which are data constants taking no arguments. This information is interpreted to mean that bitstrings representing the coverage of the

boolean type *Bool* must be of size 2 (because *Bool* has two data constructors), and each bit in the bitstring cannot be an expanded bitstring (because the data constructors are both nullary constants). Similarly, the third equation $expand\ Nat = [unit, Nat]$ means that the natural number type *Nat* has two data constructors, where the first data constructor *Zero* is a non-expandable nullary constant, and the second *Succ* is a unary data constructor whose argument is again an expandable type *Nat*. Note, the equations of *expand* exactly reflect the actual Haskell data type definitions for each of the types *Bool*, *Color*, *Nat*, *Maybe*, and *List* (excluding only their data constructor names). We will explain further details of the *expand* function and other definitions including *Type* product (\otimes) in Figure 3, when we discuss the tensor product in §3.1.4.

3.1.3 Bitwise-and ($\&$) and bitwise-or (\mid)

The definitions for bitwise-and ($\&$) and bitwise-or (\mid) are straightforward and self-explanatory. We assume both arguments of ($\&$) and (\mid) be well-formed and of the same *Type*, although we do not explicitly check them in Figure 2. We can, of course, alter the definition to test well-formedness and *Type* equality on every call to ($\&$) and (\mid), but this will incur heavy runtime overhead since ($\&$) and (\mid) invoke themselves recursively. In §4, we improve our implementation of ($\&$) and (\mid) to prevent such incompatible inputs using the type system of Haskell to check these invariants statically.

3.1.4 Tensor product (\otimes) and Type product (\otimes)

We use tensor products to calculate the coverage of tuple patterns. The coverage for a tuple pattern is the tensor product of the coverage of its components. For example, $\llbracket (True, Zero) \rrbracket = \llbracket True \rrbracket \otimes \llbracket Zero \rrbracket$. While the tensor product rule tells us how to construct the bitstring information inside *BIT*, we also need a function to construct the *Type* information stored in *BIT*. This is the role of the *Type* product (\otimes). The result of this product is often a tuple type, but in several degenerate cases (e.g., when either of the arguments is the empty tuple type *unit*) the tuple type is not needed. Here are some examples of *Type* products: $Bool \otimes Nat = Tuple\ [Bool, Nat]$, $unit \otimes Nat = Nat$, and $Bool \otimes unit = Bool$. Note, we represent *Type* information for tuple types using the data constructor *Tuple*. For example, $Tuple\ [Bool, Bool]$ represents the boolean pair type (*Bool*, *Bool*) and $Tuple\ [Bool, Bool, Bool]$ represents the boolean triple type (*Bool*, *Bool*, *Bool*). We do not allow singleton tuples of the form $Tuple\ [t]$ since functional languages do not have singleton tuple types. The empty tuple type $Tuple\ []$ is also junk (Burstall and Goguen 1981), but we give it a mnemonic name *unit* and use this (otherwise unused *BIT* value) to indicate a nullary constructor.

The *Type* product (\otimes) in Figure 3 reflects the abstract behavior of the tensor product (\otimes) in Figure 2. It computes the type and shape of the result from the types of the inputs. The relation between \otimes and \otimes is evident in the first, second, third, and the last equation in the definition of the tensor product.

The first, second, and third equation show that the behavior of tensor product is similar to bitwise-and ($\&$) when either of the arguments is 0 or when both arguments are 1. The difference is that the tensor product is well defined for any two *BIT*'s of possibly different *Types*, and its resulting *Type* is the *Type* product of the argument *Types*. For example, $I\ Bool \otimes I\ Nat = I\ (Bool \otimes Nat) = I\ (Tuple\ [Bool, Nat])$.

We emphasize that non-expandable *Types* like *unit* are the identities for the *Type* product and values like $I\ unit$ are identities for the tensor product. For example, $I\ unit \otimes I\ Nat = I\ (unit \otimes Nat) = I\ Nat$. We can see that, in the definition of (\otimes), the result of *Type* product is identical to one of its arguments when the other argument is non-expandable (i.e., expands to an empty list), and, in the sixth equation of *expand*, it is indeed

the case that the empty tuple type *unit* expands to an empty list (i.e., non-expandable). There are non-expandable *Types* other than *unit*, such as $Tuple\ [unit, unit]$, and we will see why such *Types* are non-expandable when we explain the last equation of *expand* at the end of this subsection. We exploit this identity property in the fourth and fifth equation of the tensor product definition.

The fourth and fifth equation of (\otimes) defines the behavior of the tensor product when either of the arguments is in the form of a one bit and the other is the form a bitstring. In the fourth equation, for example, when the second argument is $I\ unit$, that is when $u = unit$, it takes the first case branch $[] \rightarrow Bs\ xs\ t$ since *unit* is non-expandable. We could have equally defined the case branch as $[] \rightarrow Bs\ xs\ (t \otimes u)$, but we exploit the fact that non-expandable types like *unit* are an identities for the *Type* product (\otimes).

For expandable types, The second branch is taken. For any expandable type u , $I\ u$ expands to $Bs\ (map\ I\ us)\ u$, which is a bitstring of all 1 bits. The exact number of bits ($us = expand\ u$) is computed by using the *expand* function which knows the number of constructors of the expandable type u .

The fifth equation of (\otimes) behaves similarly to the fourth. Note, in both the fourth and fifth equation, *Type* information guides the calculation of bits. This aspect distinguishes tensor product from the other *BIT* operations in §3.1.3 where *Types* are just additional information that may help when checking input compatibility.

The last equation of (\otimes) is the workhorse of the tensor product. When both arguments are in the form of bitstrings, it performs element-wise product defined by the local operator (\otimes), which corresponds to the usual tensor product definition from mathematics (a generalization of Equation 1 from §2.4). So, when the first argument is a bitstring of length n and the second is of length m , the result of the tensor product is an $n \times m$ matrix of bits. More precisely, it is a bitstring of size n (i.e., matrix represented as n rows) whose elements are bitstrings of size m (i.e., each rows has m bits). For example, we can calculate $\llbracket (True, False) \rrbracket$ in §2.4 as follows:

$$\begin{aligned} \llbracket (True, False) \rrbracket &= \llbracket True \rrbracket \otimes \llbracket False \rrbracket \\ &= Bs\ [0\ unit, I\ unit]\ Bool \otimes Bs\ [I\ unit, 0\ unit]\ Bool \\ &= Bs\ [Bs\ [0\ unit, 0\ unit]\ Bool, \\ &\quad Bs\ [I\ unit, 0\ unit]\ Bool]\ (Bool \otimes Bool) \end{aligned}$$

Similarly, we can calculate $\llbracket (Succ\ _, Zero) \rrbracket$ as follows:

$$\begin{aligned} \llbracket (Succ\ _, Zero) \rrbracket &= \llbracket Succ\ _ \rrbracket \otimes \llbracket Zero \rrbracket \\ &= Bs\ [0\ unit, I\ Nat]\ Nat \otimes Bs\ [I\ unit, 0\ Nat]\ Nat \\ &= Bs\ [Bs\ [0\ unit, 0\ Nat]\ Nat, \\ &\quad Bs\ [I\ Nat, 0\ (Nat \otimes Nat)]\ (Nat \otimes Nat)]\ (Nat \otimes Nat) \end{aligned}$$

Here, the coverage of (*True*, *False*) and (*Succ* $_$, *Zero*) both have the common shape of a 2×2 matrix $\begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$, but they are indexed by different *Types*. Note how each 1 and 0 in $\begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$ has its own type index, and that these types differ between the two coverage bitmaps.

Finally we know enough to motivate the last equation of *expand*, which defines how non-empty tuples expand. When the first component t is a non-expandable type (e.g., *unit* or $Tuple\ []$) which only arise from nullary constructors) the first case branch is taken: $[] \rightarrow expand\ (Tuple\ xs)$. Here, we ignore the first component and expand the tuple with the rest of the components. For example,

$$\begin{aligned} expand\ (Tuple\ [unit, Bool]) &= expand\ (Tuple\ [Bool]) \\ &= expand\ Bool \\ &= [unit, unit] \end{aligned}$$

In addition, a tuple type is non-expandable when all components are non-expandable. For example,

$$\begin{aligned} \text{expand } (\text{Tuple } [unit, unit]) &= \text{expand } (\text{Tuple } [unit]) \\ &= \text{expand } unit \\ &= [] \end{aligned}$$

When the first component t is an expandable type, that is when $\text{expand } t = [t_1, \dots, t_n]$, the second case branch is taken: $ts \rightarrow [\text{foldr } (\otimes) unit (t : xs) \mid t \leftarrow ts]$. Unfolding the foldr , it is easier to see what is happening.

$$\begin{aligned} \text{expand } (t \otimes u_1 \otimes \dots \otimes u_m) &= [t_1 \otimes u_1 \otimes \dots \otimes u_m, \\ &\quad \vdots \\ &\quad t_n \otimes u_1 \otimes \dots \otimes u_m] \end{aligned}$$

This means that the coverage for tuple patterns of type (t, u_1, \dots, u_m) expands just as if the coverage for its first component expands alone, but the *Type* index for each row in the expanded bitstring is the type product of each t_i and the other components u_1, \dots, u_m . For example,

$$\begin{aligned} \text{expand } (\text{Color } \otimes \text{Bool } \otimes \text{Nat}) &= [\text{unit } \otimes \text{Bool } \otimes \text{Nat}, \\ &\quad \text{unit } \otimes \text{Bool } \otimes \text{Nat}, \\ &\quad \text{unit } \otimes \text{Bool } \otimes \text{Nat}] \\ &= [\text{Bool } \otimes \text{Nat}, \text{Bool } \otimes \text{Nat}, \text{Bool } \otimes \text{Nat}] \end{aligned}$$

3.1.5 Reducing BITs to canonical forms with *reduce*

For any $t :: \text{Type}$ of more than one data constructors, the properties $\mathbf{I } t = Bs [\mathbf{I } t_1, \dots, \mathbf{I } t_n]$ and $\mathbf{0 } t = Bs [\mathbf{0 } t_1, \dots, \mathbf{0 } t_n]$ hold, where n is the number of data constructors of the type which t represents. More precisely, these properties hold when the equation $\text{expand } t = [t_1, \dots, t_n]$ holds. The function *reduce* in Figure 2 reduces BITs into their canonical forms based on these properties. We call equivalence classes of BITs, with respect to their canonical forms, *covers*. Thus, we can read $\llbracket p \rrbracket$ as the cover of p . We will often use the word “cover” loosely, in place of “a representation of cover”.

We use *reduce* to trim down the final result or intermediate results of the computation of covers. So, it is important to ensure that *reduce* does not change the meaning of the BIT operations. For example, following properties must hold for tensor products:

$$\begin{aligned} x \otimes y &\equiv \text{reduce } x \otimes \text{reduce } y \\ x \otimes y &\equiv x \otimes \text{reduce } y \\ x \otimes y &\equiv \text{reduce } x \otimes y \end{aligned}$$

where (\equiv) defined as $x \equiv y = \text{reduce } x == \text{reduce } y$, provided that BIT and *Type* derive the *Eq* class. Similar properties must hold for bitwise-and ($\&$) and bitwise-or (\cdot) as well. We used QuickCheck (Claessen and Hughes 2000) to convince ourselves that such properties hold for well-formed inputs, as in Appendix B.

3.2 Using sparse bitmaps for pattern match coverage

Using the BIT data type and its operations, we can define basic combinators for the covers of the patterns of type *Bool*, *Maybe a*, *Nat*, $[a]$, and the wildcard pattern:

$$\begin{aligned} f &= Bs [\mathbf{I } unit, \mathbf{0 } unit] \text{Bool} && \text{-- } \llbracket \text{False} \rrbracket \\ t &= Bs [\mathbf{0 } unit, \mathbf{I } unit] \text{Bool} && \text{-- } \llbracket \text{True} \rrbracket \\ n &= \lambda a \rightarrow Bs [\mathbf{I } unit, \mathbf{0 } a] (\text{Maybe } a) && \text{-- } \llbracket \text{Nothing} \rrbracket \\ j \ x &= Bs [\mathbf{0 } unit, x] (\text{Maybe } (\text{typeof } x)) && \text{-- } \llbracket \text{Just } x \rrbracket \\ z &= Bs [\mathbf{I } unit, \mathbf{0 } \text{Nat}] \text{Nat} && \text{-- } \llbracket \text{Zero} \rrbracket \\ s \ n &= Bs [\mathbf{0 } unit, n] \text{Nat} && \text{-- } \llbracket \text{Succ } n \rrbracket \\ nil &= \lambda a \rightarrow Bs [\mathbf{I } unit, \mathbf{0 } (\text{List } a)] (\text{List } a) && \text{-- } \llbracket [] \rrbracket \\ cons \ x \ xs &= Bs [\mathbf{0 } unit, x \otimes y] (\text{List } (\text{typeof } x)) && \text{-- } \llbracket x : xs \rrbracket \\ wc &= \lambda a \rightarrow \mathbf{I } a && \text{-- } \llbracket - \rrbracket \end{aligned}$$

Then, we can build up BIT representations for the covers of the patterns in Figure 1 using the combinators defined above:

$$\begin{aligned} b1 &= n (\text{Nat } \otimes \text{List } \text{Bool}) \\ b2 &= j (wc \ \text{Nat} \quad \otimes \ \text{nil } \text{Bool}) \\ b3 &= j (z \quad \otimes \ wc (\text{List } \text{Bool})) \\ b4 &= j (s \ z \quad \otimes \ cons \ f (wc (\text{List } \text{Bool}))) \\ b5 &= j (s (wc \ \text{Nat}) \quad \otimes \ cons \ t (wc (\text{List } \text{Bool}))) \\ b6 &= j (s (s (wc \ \text{Nat})) \quad \otimes \ cons \ f (wc (\text{List } \text{Bool}))) \end{aligned}$$

Note, the covers evaluate to BITs of different depth of nesting:

$$\begin{aligned} > \text{printB } b1 && \text{-- } \llbracket N \rrbracket \\ 10 && \\ > \text{printB } b2 && \text{-- } \llbracket J (-, []) \rrbracket \\ 0[[10][10]] && \\ > \text{printB } b3 && \text{-- } \llbracket J (Z, -) \rrbracket \\ 0[[11][00]] && \\ > \text{printB } b4 && \text{-- } \llbracket J (S \ Z, F : -) \rrbracket \\ 0[[00][0[[11][00]][00]]] && \\ > \text{printB } b5 && \text{-- } \llbracket J (S \ -, T : -) \rrbracket \\ 0[[00][0[[00][11]][[00][00]][[11][11]]]] && \\ > \text{printB } b6 && \text{-- } \llbracket J (S (S \ -), F : -) \rrbracket \\ 0[[00][0[[00][[11][11]][[00][00]]]] && \end{aligned}$$

We use our own *printB* function to print the content of BITs without the *Type* information, because it becomes too verbose printing all *Type* information using the standard library *print* function, provided that BIT and *Type* derive the *Show* class. For example, even $b1$ does not print in one line:

$$\begin{aligned} > \text{print } b1 \\ Bs [\mathbf{I } (\text{Tuple } []), \mathbf{0 } (\text{Tuple } [\text{Nat}, \text{List } \text{Bool}])] \\ \quad (\text{Maybe } (\text{Tuple } [\text{Nat}, \text{List } \text{Bool}])) \end{aligned}$$

Although the covers differ in their depth of nesting, the bitwise-or operation adjusts the result with the most deeply nested one:

$$\begin{aligned} > \text{printB } (b1 \cdot b2) \\ 1[[10][10]] \\ > \text{printB } (b1 \cdot b2 \cdot b3 \cdot b4 \cdot b5) \\ 1[[11][1[[11][11]][[00][00]][[11][11]]]] \end{aligned}$$

Finally, we can see that the six patterns are exhaustive by applying *reduce* to the result of the bitwise-or operation over the six covers:

$$\begin{aligned} > \text{printB } (b1 \cdot b2 \cdot b3 \cdot b4 \cdot b5 \cdot b6) \\ 1[[11][1[[11][11]][[11][11]][[11][11]]]] \\ > \text{printB } (\text{reduce } (b1 \cdot b2 \cdot b3 \cdot b4 \cdot b5 \cdot b6)) \\ 1 \end{aligned}$$

In addition, we can also see that $b2$ and $b3$ overlap but others don't, by performing bitwise-and operation on them:

$$\begin{aligned} > \text{printB } (\text{reduce } (b2 \ \& \ b3)) \\ 0[[10][0]] \\ > \text{printB } (\text{reduce } (b1 \ \& \ b2)) \\ 0 \\ > \text{printB } (\text{reduce } (b3 \ \& \ b5)) \\ 0 \end{aligned}$$

4. Towards type safe and polymorphic BITs

The sparse bitmap implementation in the previous section leaves room for improvement. First, the BIT datatype, in Figure 2, admits junk (Burstall and Goguen 1981). That is, we can define well-typed but ill-formed covers that does not represent any real pattern. For

example, $Bs [I \text{ Nat}, 0 \text{ Color}, I \text{ Bool}] \text{ Bool}$ is a well-typed *BIT* but makes little sense since *Bool* only has two nullary data constants. Second, we can only construct covers for mono-morphically typed patterns. Although we have parameterized *Types* such as *Maybe* and *List* in Figure 3, we can only handle specific instances because we do not have value representation for type variables in the *Type* datatype. This explains why *wc* (the wildcard pattern $_$) and *n* (*Nothing*) in §3.2 take explicit *Type* arguments.

We could add an additional data constructor for type variables in Figure 3 (e.g. `data Type = Var Name | ...`), and manually keep track of the type environment and unification of the type variables during the *BIT* calculation.⁴ But it would be preferable if the mechanism of the underlying type system handled type variables, as in typed embedded languages (Pašalić et al. 2002; Xi et al. 2003). In this section we use advanced features of Haskell (with some GHC extensions) to try and reach this goal. We view the uses of these features as the second part of the pearl, as we believe the approach we take is widely applicable to similar problems.

Unfortunately, we had only partial success implementing a type safe and polymorphic version of the *BIT* data structure and its operations in Haskell. Our approach succeeds until we need to define the tensor product. Part of the pearl is our scrutiny of what makes it difficult to implement tensor product, and some thoughts on what improvements within the current Haskell implementation might allow us to succeed.

4.1 Type safe and polymorphic BITs

We can define type safe *BITs* which can handle patterns of polymorphic types using Generalized Algebraic Data Types (GADTs) (Fig. 4) and type families (a.k.a. open type functions) (Fig. 5).

In Figure 4, the datatype $BIT\ t$ is indexed by a type t . The indexing is accomplished by adding a type parameter to the data declaration to *BIT*. This is in contrast to the adding an embedded value level *Type* as a component to each of the constructors of *BIT* in Figure 2

In Figure 4, $0 :: BIT\ t$ and $I :: BIT\ t$ are polymorphic bits that represent the empty cover and the full cover for any type t . $Bs :: BList\ (Expand\ t) \rightarrow BIT\ t$ takes a well-formed bit list for type t and constructs a bit for that type.

Note that $Blist\ t$ is structurally identical to $[Bit\ a]$, but the index t tracks well formedness of the list rather than the type of elements in the list. The $Expand\ t$ in the type of Bs is how we ensure the list is well formed. We use the uninhabited data types $Nil :: *$ and $(:.) :: * \rightarrow * \rightarrow *$ to build *type level* lists, t , which we use as type indexes for $BList\ t$. Nil is the empty type level list, and $t : ts$ is a non-empty type level list whose head and tail are t and ts . For example, $() : Nat : Nil$ is a type level list of length 2 whose first and second elements are $()$ and Nat .

The data type $BList\ ts$ defines type-indexed bit lists. $BNil$ is the empty bit list whose type is indexed with the empty type level list Nil . Given a bit $b :: BIT\ t$ and a bit list $bs :: BList\ ts$, we can construct a non-empty bit list $b :: bs$ whose type index is $t : ts$.

Note, Bs can only construct well-formed bits because the type index for its bit list argument is constrained to be $(Expand\ t)$. The type family $Expand$ in Figure 5 defines rules for well-formedness for each type. For example, the sixth type family instance $Expand\ Nat = () : Nat : Nil$ means that Nat has two data constructors where the first data constructor $Zero$ is a nullary constant and the second $Succ$ is a unary constructor whose argument is of type Nat . Thus, the type checker only accepts well-formed *BITs* such as $Bs\ (I :: 0 :: BNil)$, but rejects ill-formed *BITs* such as $Bs\ (I :: 0 :: I :: BNil)$.

⁴We actually need further changes than what we described here. See §4.2.

data BIT t where

$0 :: BIT\ t$

$I :: BIT\ t$

$Bs :: BList\ (Expand\ t) \rightarrow BIT\ t$

infixr 2 ::

infixr 2 ::

data Nil

data x :: xs

data BList bs where

$BNil :: BList\ Nil$

$(::) :: BIT\ t \rightarrow BList\ ts \rightarrow BList\ (t : ts)$

$(.\&) :: BIT\ t \rightarrow BIT\ t \rightarrow BIT\ t$

$0 \ .\& _ = 0$

$_ \ .\& \ 0 = 0$

$I \ .\& \ y = y$

$x \ .\& \ I = x$

$Bs\ xs \ .\& \ Bs\ ys = Bs\ (xs \ .\& \ ys)$

$(.\&.) :: BList\ ts \rightarrow BList\ ts \rightarrow BList\ ts$

$(.\&.) = zipBWith\ (.\&)$

$(.|) :: BIT\ t \rightarrow BIT\ t \rightarrow BIT\ t$

$I \ .| \ _ = I$

$_ \ .| \ I = I$

$0 \ .| \ y = y$

$x \ .| \ 0 = x$

$Bs\ xs \ .| \ Bs\ ys = Bs\ (xs \ .| \ ys)$

$(.|) :: BList\ ts \rightarrow BList\ ts \rightarrow BList\ ts$

$(.|) = zipBWith\ (|)$

$reduce :: BIT\ t \rightarrow BIT\ t$

$reduce\ (Bs\ xs) = \text{case } reduceBs\ xs \text{ of}$

$0 :: xs \mid \text{allO } xs \rightarrow 0$

$I :: xs \mid \text{allI } xs \rightarrow I$

$xs' \rightarrow Bs\ xs'$

$reduce\ x = x$

$reduceBs :: BList\ ts \rightarrow BList\ ts$

$reduceBs\ BNil = BNil$

$reduceBs\ (x :: xs) = reduce\ x :: reduceBs\ xs$

$zipBWith :: (\forall t . BIT\ t \rightarrow BIT\ t \rightarrow BIT\ t)$

$\rightarrow BList\ xs \rightarrow BList\ xs \rightarrow BList\ xs$

$zipBWith\ f\ BNil\ BNil = BNil$

$zipBWith\ f\ (x :: xs)\ (y :: ys) = f\ x\ y :: zipBWith\ f\ xs\ ys$

$allO :: BList\ ts \rightarrow Bool$

$allO\ BNil = True$

$allO\ (0 :: xs) = allO\ xs$

$allO\ (_ :: xs) = False$

$allI :: BList\ ts \rightarrow Bool$

$allI\ BNil = True$

$allI\ (I :: xs) = allI\ xs$

$allI\ (_ :: xs) = False$

Figure 4. Type safe and polymorphic *BIT* and its operations: bitwise-or($\cdot|$), bitwise-and($\cdot\&$), and *reduce*.

```

type family   Expand t
type instance Expand ()       = Expand Nil
type instance Expand (a, b)   = Expand (a :: b :: Nil)
type instance Expand Bool     = () :: () :: Nil
type instance Expand (Maybe t) = () :: t :: Nil
type instance Expand Nat      = () :: Nat :: Nil
type instance Expand [t]      = () :: (t, [t]) :: Nil
type instance Expand Nil      = Nil
type instance Expand (t :: ts) = ...
:

```

Figure 5. Type family *Expand*

We can define basic combinators for the covers of the patterns of type *Bool*, *Maybe a*, *Nat*, and the wildcard pattern, as in §3.2:

```

f :: BIT Bool           -- [[False]]
f = Bs (I :: 0 :: BNil)
t :: BIT Bool           -- [[True]]
t = Bs (0 :: I :: BNil)
n :: BIT (Maybe t)    -- [[Nothing]]
n = Bs (I :: 0 :: BNil)
j :: BIT t → BIT (Maybe t) -- [[Just x]]
j x = Bs (0 :: x :: BNil)
wc = I                 -- [[_]]

```

Here, *n*, *j* and *wc* have truly polymorphic types. For example,

```

> :t wc
wc :: BIT t
> :t n
n :: BIT (Maybe t)
> :t (j n)
j (j wc) :: BIT (Maybe (Maybe t))
> :t (j (j wc))
j (j wc) :: BIT (Maybe (Maybe t))

```

We can evaluate these polymorphic *BIT*s or apply bitwise operations to them as follows:

```

> printB wc
1
> printB n
10
> printB (j n)
0[10]
> printB (j (j wc))
0[01]
> printB (n .| j n .| j (j wc))
1[11]
> printB (reduce (n .| j n .| j (j wc)))
1

```

Operations on bits of incompatible types now cause type errors:

```

> t .| n
Could not match expected type Bool against
inferred type Maybe t
Expected type: BIT Bool
Inferred type: BIT (Maybe t)

```

In the second argument of *(.)* namely *n*
In the expression: *t .| n*

4.2 Problems with tensor product

Recall that the tensor product is a type directed computation. In particular, we may expand a single one-bit to a bitstring of all ones during a computation. The size and shape of the “all ones” depends on the type of the original bit. The standard way of expressing such type directed computation in Haskell is via type classes. Information which was once explicit (in the *Type* components of *BIT*s) is now implicit, and must be inferred. In Haskell, implicit type directed computation is implemented using classes and instances. We can define a type class *Expandable* and its instances as follows:

```

class Expandable t where
  expand :: BIT t → BList (Expand t)
instance Expandable () where
  expand I = BNil
instance Expandable Bool where
  expand I = I :: I :: BNil
:

```

We tried many approaches, none of which were successful. The problem is captured in the program fragment below where we attempt to define the tensor product using the *Expandable* type class and its method *expand*:

```

infixr 7 ⊗
(⊗) :: (Expandable t, Expandable u) ⇒
      BIT t → BIT u → BIT (t ⊗ u)
_           ⊗ 0           = 0
0           ⊗ _           = 0
I           ⊗ I           = I
x@(Bs xs)  ⊗ (y@I :: BIT u) =
case expand y of
  BNil → x -- type error t ⊗ u ~ t
  ys   → x ⊗ (Bs ys :: BIT u)
(x@I :: BIT t) ⊗ y@(Bs ys) =
case expand x of
  BNil → y -- type error t ⊗ u ~ u
  xs   → (Bs xs :: BIT t) ⊗ y
(Bs xs :: BIT t) ⊗ (Bs ys :: BIT u) = ...

```

The problems arise in the case arms labeled -- type error. We summarize three issues which we found making it difficult to define a type safe tensor product that could handle bits of polymorphic types.

First, the definition for type level product \otimes gets very lengthy. In a type family definition we must enumerate all cases without overlapping because we do not have if-then-else or case expressions at the type level.

```

infixr 7 ⊗
type family   t ⊗ u
type instance () ⊗ u       = u
type instance Bool ⊗ ()    = Bool
type instance Bool ⊗ Bool  = Bool :: Bool :: Nil
type instance Bool ⊗ Maybe u = Bool :: Maybe u :: Nil
type instance Bool ⊗ Nat   = Bool :: Nat :: Nil
type instance Bool ⊗ [u]   = Bool :: [u] :: Nil
type instance Bool ⊗ (u :: us) = Bool :: (u :: us) :: Nil
:

```

One needs $O(n^2)$ number of instances for n types. This is really just an inconvenience, but illustrates the lack of maturity in new language features. The following two problems are much more crucial ones.

Second, we get type errors in the case branches labelled `-- type error`. To convince the type system that this definition is indeed correct, we need a mechanism to inform the type system about the following property on the identity for type product (\otimes):

$$\text{Expand } t \sim \text{Nil} \text{ implies both } t \otimes u \sim u \text{ and } u \otimes t \sim u$$

This property is a compound type invariant based on the three basic type invariants (1) $\text{Expand } t \sim \text{Nil}$, (2) $t \otimes u \sim u$, and (3) $u \otimes t \sim u$. Type invariants for Haskell have been recently proposed (Schrijvers et al. 2009) as an extension to Haskell, and have not yet been implemented in GHC 6.10. It is unclear if their proposal is the best way to proceed, but it would solve our problem.

Third, even if we have overcome the previous two issues, we still cannot apply functions with type class constraints to fully polymorphic values. For example, $wc \otimes wc$ will not type check. The type system complains that ambiguous type variables appear in the class constraints. To discharge the class constraint, we must provide fixed instances of the type variables appearing in the class constraint. We know that it is okay to perform computation without fixing those types when both arguments of (\otimes) are `Is`, but the language implementations in general cannot figure out such properties.

In addition to the three issues related to type system, it would also be beneficial to continue to perform calculation as far as possible without invoking the `expand` method, exploiting the distributivity of tensor product over other bitwise-operations. For example,

$$\begin{aligned} & (wc \otimes n) \cdot (wc \otimes j \cdot wc) \\ &= (I \otimes Bs (I :: 0 :: Nil)) \cdot (I \otimes Bs (0 :: I :: Nil)) \\ &= I \otimes (Bs (I :: 0 :: Nil) \cdot Bs (0 :: I :: Nil)) \\ &= I \otimes Bs (I :: I :: Nil) \quad \text{-- applying } reduce \text{ to lhs} \\ &= I \otimes I = I \end{aligned}$$

Programming languages usually do not support computations following such custom evaluation strategies. One possible way of dealing with this problem would be introducing new forms of data to express such suspended computation when either of the argument is `I` and the other is a bit list. For example, we might add two more constructors to `BIT` as follows:

```
data BIT t where
  0  :: BIT t
  I  :: BIT t
  Bs :: BList (Expand t) -> BIT t
  IX :: BList (Expand u) -> BIT (t ⊗ u) -- I ⊗ Bs ys
  XI :: BList (Expand t) -> BIT (t ⊗ u) -- Bs xs ⊗ I
```

and also add more equations for other bitwise-operations to handle these two new constructors. Note, this evaluation strategy not solve any of the three typing issues we previously discussed, because the type system will still require instantiations of type variables to discharge type class constraints. For example, polymorphic expressions such as $wc \otimes wc$ will not start to compute, even if we had such a clever evaluation strategy.

5. Related Work

Maranget (2007) tackles the pattern match coverage problem from the point of specializing the decision tree computational algorithm, but realizes that his work turns out to be more general than just a strip down version of the pattern compilation algorithm. We take a further step beyond by considering the pattern match coverage problem in isolation from the beginning, and focus on giving nat-

ural, compact, and composable representations for the coverage of patterns.

Krishnaswami (2009) gives a novel coverage checking algorithm in the context of mechanizing proof for correctness of pattern compilation strategy. He defines wildcard (\top), and-patterns ($p_1 \wedge p_2$), or-patterns ($p_1 \vee p_2$), false-pattern (\perp), and complement patterns ($\neg p$) in his language. Then, he defines proof rules for coverage judgments over those patterns. His minimal language is well-suited for deterministic proof on the correctness of pattern matching coverage and compilation algorithms. For the sake of illustrating rigorous proofs, the language in his paper only includes product types ($A \times B$) and sum types ($A + B$), although he mentions that his framework can easily be extended for iso-recursive types ($\mu\alpha.A$). In contrast, we address concrete implementation issues that are applicable to functional languages with algebraic data types. We specify the properties of algebraic data types and give data representation of pattern coverage. One can also view our work as giving a concrete computational model for such proof theoretic approaches.

Other researchers and practitioners using Haskell also have experienced subtleties in encoding polymorphic values and types, as we have found it difficult to represent covers of polymorphic patterns or defined type safe version of the tensor product. One known trick to encode polymorphic types using the host language type system is using type level de Bruijn indices (Guillemette and Monnier 2008), but we are not yet sure whether this will help solve our problems. In the Haskell-Cafe mailing lists, there was a discussion on encoding polymorphic types into data, which is more relevant to our problem. Lennart Augustsson (2009-03-25) opened a thread asking for tricks or language extensions that can make $\text{typeOf} :: (\text{Typeable } a) \Rightarrow a \rightarrow \text{TypeRep}$ applicable to polymorphic values such as $\text{id} :: a \rightarrow a$. Lennart commented in his posting that this question gets more interesting as GHC moves towards first class polymorphic values. The application typeOf id fails to type check because the variable a in the class constraint ($\text{Typeable } a$) does not become fully instantiated, which is exactly the same reason for our problem that $wc \otimes wc$ fails to type check in §4.2. Oleg Kiselyov (2009-03-26), in response to Lennart, suggested a solution based on similar techniques used in his “Reversing Haskell typechecker” (Kiselyov 2005-05-01). Although the code included in Oleg’s posting has some bugs (e.g., evaluating $\text{mytypof } (\text{Just } [])$ shows Maybe any1 rather than $\text{Maybe } [anyI]$), it is worth looking into it for insights.

Dependently typed languages (Norell 2009; McBride 2004) are known to be suitable for expressing type directed computation naturally. However, most well known examples are type collapsing operations (e.g., generic serialization), type preserving transformations (e.g., typed embedded interpreters), or operations on values with scalar indices (e.g., appending vectors). It would be interesting to see whether dependently typed languages can help us express type safe and polymorphic sparse bitmaps more easily.

6. Conclusions and Future Work

We have presented a pattern match coverage checking algorithm based on simple bitwise-or operation, interpreting each pattern as a generalized bitstring. Our sparse bitmap data structure implements this bitstring semantics of patterns in a compact and flexible way. An innovative aspect of our approach is that we have a compact representation of non-exhaustive covers, which we can compose incrementally or even store as data for later use (e.g., compiler backend, program analysis, linting tools). To integrate our work into practical tools like Catch (Mitchell and Runciman 2008), we would need to generate the specification of algebraic data types (the `expand` function in Fig. 2) and basic combinators (e.g., wc , n , j , z , s in §3.2) automatically from the source code.

The limitation of our current work is that we can only calculate sparse bitmap representation for specific instances of a polymorphic type, but cannot calculate bitmaps for the polymorphic type itself in general. Note, this does not mean we fail to give definitive answers for the exhaustiveness of patterns used in polymorphic function definitions. We can definitely check exhaustiveness of a fixed set of polymorphic patterns by instantiating the type variables in the polymorphic type to arbitrary types of our choice (preferably to the most simple one like *unit*). What this limitation really means is that we have not yet clearly identified how to represent non-exhaustive covers of polymorphic types, in order to compose them incrementally or store them as data for later use.

During our exploration towards type safe and polymorphic sparse bitmap, we have identified several interesting research questions for future work. First, what features are necessary for host languages to encode typed embedded languages with polymorphic values and type directed operations? Second, how can we specify and implement custom evaluation strategies based on common algebraic laws (e.g., distributivity of \otimes over $\cdot|$ and $\cdot&$), especially when there are type directed computations (e.g., \otimes)? We think our sparse bitmap structure is a very interesting case study problem because there are both type preserving computations (bitwise-or ($\cdot|$), bitwise-and ($\cdot&$), and *reduce*) and type directed computations (tensor product (\otimes)) that require custom evaluation strategies. More interestingly, the type product (\otimes) forms a non-trivial algebra with multiple identities and there is a type invariant relating two type functions (\otimes) and *Expand*.

Acknowledgments

Thanks to Mark Jones and Chuan-kai Lin for proofreading the draft, Tim Chevalier for the help on using QuickCheck, and Tom Harche and other members of Computer Science Graduate Student Council and the PSU Association for Computing Machinery for organizing Friday talks at Portland State University.

References

- Lennart Augustsson. Compiling pattern matching. In *Proceedings of the Second Conference on Functional Programming Languages and Computer Architecture*, pages 368–381, 1985.
- Lennart Augustsson. typeof for polymorphic value, 2009-03-25. URL <http://article.gmane.org/gmane.comp.lang.haskell.cafe/55738>.
- Rod Burstall and Joseph Goguen. An informal introduction to specifications using Clear. In *The Correctness Problem in Computer Science*, pages 185–213. Academic, 1981. Reprinted in *Software Specification Techniques*, Narain Gehani and Andrew McGettrick, editors, Addison-Wesley, 1985, pages 363–390.
- Luca Cardelli. Compiling a functional language. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 208–217, 1984.
- Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *ICFP*, pages 268–279, 2000.
- Louis-Julien Guillemette and Stefan Monnier. A type-preserving compiler in haskell. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 75–86, 2008.
- M. Hanus. *Report on Curry (ver.0.8.2)*. Inst. fur Informatik, Christian-Albrechts Universitat, .de, 2006.
- Oleg Kiselyov. typeof for polymorphic value, 2009-03-26. URL <http://article.gmane.org/gmane.comp.lang.haskell.cafe/55744>.
- Oleg Kiselyov. Reversing haskell typechecker: converting from undefined to defined (version 1.1), 2005-05-01. URL <http://okmij.org/ftp/Haskell/types.html#de-typechecker>.

- Neelakantan R. Krishnaswami. Focusing on pattern matching. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 366–378, 2009.
- Serge Lang. *Algebra*, volume 211 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, third edition, 2002.
- Luc Maranget. Warnings for pattern matching. *J. Funct. Program*, 17(3): 387–421, 2007.
- Conor McBride. Epigram: Practical programming with dependent types. In *Advanced Functional Programming*, volume 3622 of *LNCS*, pages 130–170. Springer, 2004.
- Neil Mitchell and Colin Runciman. Not all patterns, but enough: an automatic verifier for partial but sufficient pattern matching. In *Haskell '08: Proceedings of the first ACM SIGPLAN Symposium on Haskell*, pages 49–60, 2008.
- Ulf Norell. Dependently typed programming in Agda. In *Proceedings of TLDI'08: 2008 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, pages 1–2, 2009.
- Emir Pašalić, Walid Taha, and Tim Sheard. Tagless staged interpreters for typed languages. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN International Conference on Functional Programming*, pages 218–229, 2002.
- Mikael Pettersson. A term pattern-match compiler inspired by finite automata theory. In *CC '92: Proceedings of the 4th International Conference on Compiler Construction*, pages 258–270, 1992.
- Simon Peyton Jones et al., editors. *Haskell 98 Language and Libraries, the Revised Report*. Cambridge University Press, 2003.
- Tom Schrijvers, Louis-Julien Guillemette, and Stefan Monnier. Type invariants for haskell. In *PLPV '09: Proceedings of the 3rd workshop on Programming languages meets program verification*, pages 39–48, 2009.
- Hongwei Xi, Chiyang Chen, and Gang Chen. Guarded recursive datatype constructors. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 224–235, 2003.

A. Expanding types with single data constructor

We decided to avoid examples involving data types with single data constructor such as $()$ in the main sections of the paper, because of some implementation details of *expand* over these types may cause confusion, although they are just implementation trivia. So, we discuss how to define *expand* function over types with single data constructors such as $\mathbf{data} () = ()$ in Haskell, here in the appendix.

The Haskell unit type $()$ is not the same as *unit* in the paper (which stand for non expandable data constants like *True*) although their *BIT* representation may be isomorphic. Note, *True* has type *Bool* not $()$. Thus, we should define an additional data constructor $\mathit{Unit} :: \mathit{Type}$ in Figure 3 to represent $()$, as follows:

$$\mathbf{data} \mathit{Type} = \mathit{Unit} \mid \mathit{Bool} \mid \mathit{Color} \mid \mathit{Nat} \\ \mid \mathit{Maybe} \mathit{Type} \mid \mathit{List} \mathit{Type} \mid \mathit{Tuple} [\mathit{Type}]$$

Then, we have two possible choices of defining *expand* on Unit :

$$\mathit{expand} \mathit{Unit} = [\mathit{unit}] \\ \mathit{expand} \mathit{Unit} = [] \quad \text{-- or } \mathit{expand} \mathit{unit}$$

The former is indeed correct, but the latter is a more efficient optimization. This optimization is based on the observation that pattern matches for $()$ are always exhaustive, thus needless to check their exhaustiveness. Note, the only possible (non divergent) value is $() :: ()$. So, any pattern of type $()$, either $()$ or $_$, are exhaustive match by itself. Since case expressions or function definitions by pattern syntactically requires at least one pattern, all pattern matches against values of type $()$ must be exhaustive.

In general, we can define optimized versions of *expand* over data types with single data constructors. For example, consider the

data type T defined as `data T = C Bool`. To represent T as a $Type$ value, we add another data constructor $T :: Type$ as follows:

```
data Type = T | Unit | Bool | Color | Nat
          | Maybe Type | List Type | Tuple [Type]
```

Again, we have two possible choices of defining `expand` on T :

```
expand T = [Bool]
expand T = expand Bool
```

The latter is an optimization of the former based on the observation that any pattern match against values of type T will match against the constructor C , so we only need to consider its argument type.

We can also define optimized versions of `expand` when types like $()$ or T appear in other data types, which may have more than one data constructors. For example, consider the type U defined as `data U = C1 () | C2 T`. We have two possible choices of defining `expand` on U :

```
expand U = [Unit, T]
expand U = [unit, Bool]
```

The latter is an optimization of the former.

B. QuickCheck for BITs

We use QuickCheck version 2.1.0.1 on GHC version 6.10.1. We generate arbitrary $Types$ randomly by declaring $Type$ as an instance of the $Arbitrary$ class and implementing the $arbitrary$ method. We define the custom generator $bit :: Type \rightarrow Gen BIT$ that generates random BIT s of a given $Type$.

```
import BIT
import Monad
import Test.QuickCheck

instance Arbitrary Type where arbitrary = tygen
tygen :: Gen Type
tygen = sized tygen'
tygen' 0 = oneof $ map return [Bool, Nat,
                              Color, Tuple []]
tygen' n | n > 0 = oneof [liftM Maybe t, liftM List t,
                          liftM Tuple ts]
    where t = tygen' (n `div` 3)
          ts = liftM2 (:) t $ liftM2 (:) t $
              resize (n `div` 3) arbitrary

bit :: Type \rightarrow Gen BIT
bit t = sized (bit' t)
bit' t 0 = oneof $ map return [I t, O t]
bit' t n | n > 0 =
  case expand t of
    [] \rightarrow bit' t 0
    ts \rightarrow liftM Bs (sequence [bit' t k | t \leftarrow ts]) `ap`
      return t
    where k = (n - 1) `div` length ts

bits :: [Type] \rightarrow Gen [BIT]
bits = sequence \circ map bit

forAllb :: (Testable prop) \Rightarrow
  Type \rightarrow (BIT \rightarrow prop) \rightarrow Property
forAllb = forAll \circ bit

forAllbs :: (Testable prop) \Rightarrow
  [Type] \rightarrow ([BIT] \rightarrow prop) \rightarrow Property
forAllbs = forAll \circ bits
```

Then, we define properties of BIT s using the test generators above.

```
infix 4 \equiv
x \equiv y = reduce x == reduce y
prop_or_O_lhs t = forAllb t $ \lambda x \rightarrow O t .| x \equiv x
prop_or_O_rhs t = forAllb t $ \lambda x \rightarrow x .| O t \equiv x
prop_or_I_lhs t = forAllb t $ \lambda x \rightarrow I t .| x \equiv I t
prop_or_I_rhs t = forAllb t $ \lambda x \rightarrow x .| I t \equiv I t
prop_or_xx t = forAllb t $ \lambda x \rightarrow x .| x \equiv x
prop_or_commute t = forAllbs [t, t] $ \lambda [x, y] \rightarrow
  x .| y \equiv y .| x
prop_or_assoc t = forAllbs [t, t, t] $ \lambda [x, y, z] \rightarrow
  (x .| y) .| z \equiv x .| (y .| z)
prop_or_reduce_l t = forAllbs [t, t] $ \lambda [x, y] \rightarrow
  x .| y \equiv reduce x .| y
prop_or_reduce_r t = forAllbs [t, t] $ \lambda [x, y] \rightarrow
  x .| y \equiv x .| reduce y
prop_or_reduce t = forAllbs [t, t] $ \lambda [x, y] \rightarrow
  x .| y \equiv reduce x .| reduce y
prop_and_O_lhs t = forAllb t $ \lambda x \rightarrow O t .& x \equiv O t
prop_and_O_rhs t = forAllb t $ \lambda x \rightarrow x .& O t \equiv O t
prop_and_I_lhs t = forAllb t $ \lambda x \rightarrow I t .& x \equiv I t
prop_and_I_rhs t = forAllb t $ \lambda x \rightarrow x .& I t \equiv I t
prop_and_xx t = forAllb t $ \lambda x \rightarrow x .& x \equiv x
prop_and_commute t = forAllbs [t, t] $ \lambda [x, y] \rightarrow
  x .& y \equiv y .& x
prop_and_assoc t = forAllbs [t, t, t] $ \lambda [x, y, z] \rightarrow
  (x .& y) .& z \equiv x .& (y .& z)
prop_and_reduce_l t = forAllbs [t, t] $ \lambda [x, y] \rightarrow
  x .& y \equiv reduce x .& y
prop_and_reduce_r t = forAllbs [t, t] $ \lambda [x, y] \rightarrow
  x .& y \equiv x .& reduce y
prop_and_reduce t = forAllbs [t, t] $ \lambda [x, y] \rightarrow
  x .& y \equiv reduce x .& reduce y
prop_tprod_O_lhs t u = forAllb u $ \lambda y \rightarrow
  O t \otimes y \equiv O (t \otimes u)
prop_tprod_O_rhs t u = forAllb t $ \lambda x \rightarrow
  x \otimes O u \equiv O (t \otimes u)
prop_tprod_II t u = I t \otimes I u \equiv I (t \otimes u)
prop_tprod_reduce_l t u = forAllbs [t, u] $ \lambda [x, y] \rightarrow
  x \otimes y \equiv reduce x \otimes y
prop_tprod_reduce_r t u = forAllbs [t, u] $ \lambda [x, y] \rightarrow
  x \otimes y \equiv x \otimes reduce y
prop_tprod_reduce t u = forAllbs [t, u] $ \lambda [x, y] \rightarrow
  x \otimes y \equiv reduce x \otimes reduce y
prop_dist_or_and t = forAllbs [t, t, t] $ \lambda [x, y, z] \rightarrow
  x .| (y .& z) \equiv (x .| y) .& (x .| z)
prop_dist_and_or t = forAllbs [t, t, t] $ \lambda [x, y, z] \rightarrow
  x .& (y .| z) \equiv (x .& y) .| (x .& z)
prop_dist_tprod_or t u = forAllbs [t, u, u] $ \lambda [x, y, z] \rightarrow
  x \otimes (y .| z) \equiv (x \otimes y) .| (x \otimes z)
prop_dist_tprod_and t u = forAllbs [t, u, u] $ \lambda [x, y, z] \rightarrow
  x \otimes (y .& z) \equiv (x \otimes y) .& (x \otimes z)
```