# *Every Bit Counts: The binary representation of typed data and programs*

ANDREW J. KENNEDY and DIMITRIOS VYTINIOTIS
*Microsoft Research, Cambridge, CB3 0FB, UK*
(*e-mail:* {`akenn,dimitris`}`@microsoft.com`)

## Abstract

We show how the binary encoding and decoding of typed data and typed programs can be understood, programmed, and verified with the help of question-answer games. The encoding of a value is determined by the yes/no answers to a sequence of questions about that value; conversely, decoding is the interpretation of binary data as answers to the same question scheme.

We introduce a general framework for writing and verifying game-based codecs. We present games for structured, recursive, polymorphic, and indexed types, building up to a representation of well-typed terms in the simply-typed $\lambda$-calculus with polymorphic constants. The framework makes novel use of isomorphisms between types in the definition of games. The definition of isomorphisms together with additional simple properties make it easy to prove that codecs derived from games never encode two distinct values using the same code, never decode two codes to the same value, and interpret any bit sequence as a valid code for a value or as a prefix of a valid code.

## 1 Introduction

Let us play a guessing game:

I am a simply-typed program.[1] Can you guess which one?

Are you a function application? No.

You must be a function. Is your argument a `Nat`? Yes.

Is your body a variable? No.

Is your body a function application? No.

It must be a function. Is its argument a `Nat`? Yes.

Is its body a variable? Yes.

Is it bound by the nearest $\lambda$? No.

You *must be* $\lambda x{:}\texttt{Nat}.\lambda y{:}\texttt{Nat}.x$. *You're right!*

[1] A closed program in the simply-typed $\lambda$-calculus with types $\tau ::= \texttt{Nat} \mid \tau \to \tau$ and terms $e ::= x \mid e\ e \mid \lambda x{:}\tau.e$, identified up to $\alpha$-equivalence. We have deliberately impoverished the language for simplicity of presentation; in practice there would also be constants, primitive operations, and perhaps other constructs.

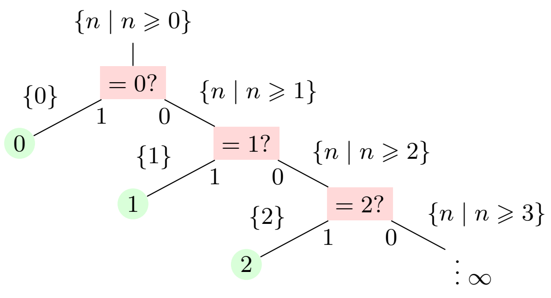
From the answer to the first question, we know that the program is not a function application. Moreover, the program is closed, and so it *must* be a $\lambda$-abstraction; hence we proceed to ask new questions about the argument type and body. We continue asking questions until we have identified the program. In this example, we asked just seven questions. Writing 1 for *yes*, and 0 for *no*, our answers were 0100110. This is a *code* for the program $\lambda x{:}\texttt{Nat}.\lambda y{:}\texttt{Nat}.x$.

By deciding a question scheme for playing our game we've thereby built an *encoder* for programs. By interpreting a bit sequence as answers to that same scheme, we have a *decoder*. Correct round-tripping of encoding and decoding follows automatically. If, as in this example, we never ask 'silly questions' that reveal no new information, then every code represents some value, or is the prefix of a valid code. In other words, *every bit counts*.

Related ideas have previously appeared in domain-specific work; tamper-proof bytecode (Franz *et al.*, 2002; Haldar *et al.*, 2002) and compact proof witnesses in proof carrying code (Necula & Rahul, 2001). In the latter case, an astonishing improvement of a factor of 30 in proof witness size is reported compared to previous syntactic representations! By contrast, standard serialization techniques do not easily guarantee tamper-proof codes, nor take advantage of semantic information to yield more compact encodings.

Our paper identifies and formalizes a key intuition behind those works: question-and-answer games. Moreover, we take a novel *typed* approach to codes, using types for domains of values, and representing the partitioning of the domain by *type isomorphisms*. Concretely, our contributions are as follows:

- We introduce question-and-answer games for encoding and decoding: a novel way to think about and program codecs (Section 2). We build codecs for numeric types, and provide combinators that construct complex games from simpler ones, producing coding schemes for structured, recursive, polymorphic, and indexed types that are correct by construction (Section 3).
- Under easily-stated assumptions concerning the structure of games, we prove round-trip properties of encoding and decoding, and the 'every bit counts' property of the title (Section 4).
- We develop more sophisticated codecs for abstract types such as sets, multisets, and permutations, making crucial use of the invariants associated with such types (Section 5), and reveal an interesting connection between Knuth's *parsimonious* algorithms and 'every bit counts' coding schemes.
- We build games for untyped and simply-typed terms that yield every-bit-counts coding schemes (Section 6). We explain how to handle polymorphic constants and sketch the extension to ML-style *let* polymorphism. Stated plainly: we can represent programs such that every sufficiently-long bit string represents a well-typed term. To our knowledge, this is the first such coding scheme for typed languages that has been proven correct.
- We show how to make use of the statistical distribution of values in our game framework by giving the concrete case for Huffman codes (Section 7). We also discuss the extension of our framework to arithmetic coding.



$\{n \mid n \geqslant 0\}$
$= 0?$
$\{0\}$
$\{n \mid n \geqslant 1\}$
1
0
0
$= 1?$
$\{1\}$
$\{n \mid n \geqslant 2\}$
1
0
1
$= 2?$
$\{2\}$
$\{n \mid n \geqslant 3\}$
1
0
2
$\infty$

Fig. 1. Unary game for naturals

- We discuss filters on games (Section 8). Finally, we discuss future developments and present connections to related work (Sections 9 and 10).

We will be using Haskell but the paper is accompanied by a partial (for reasons discussed in Section 9.3) Coq formalization, downloadable from the authors' web pages. The correctness and compactness properties of our coding schemes follow *by construction* in our Coq development, and by localized reasoning in our Haskell code. We make use of infinite structures, utilizing laziness in Haskell, and co-induction in Coq, but the techniques should adapt to a call-by-value setting through the use of thunks.

Finally, a shorter version of this paper has appeared in the proceedings of ICFP 2010. In this version we have made several presentation and restructuring modifications to include more Coq formal statements, and added new material: New number codes (Section 3), the relation to parsimonious algorithms (Section 5.3), Huffman codes and arithmetic coding (Section 7), and codes for polymorphicaly typed programs (Section 6.3).

## 2 From games to codecs

We can visualize question-and-answer games graphically as binary decision trees.

Figure 1 visualizes a (naïve) game for natural numbers. Each rectangular node contains a question, with branches to the left for *yes* and right for *no*. Circular leaf nodes contain the final result that has been determined by a sequence of questions asked on a path from the root. Arcs are labelled with the 'knowledge' at that point in the game, characterised as subsets of the original domain.

Let's dry-run the game. We start at the root knowing that we're in the set $\{n \mid n \geqslant 0\}$. First we ask whether the number *is exactly* 0 or not. If the answer is *yes* we continue on the left branch and immediately reach a leaf that tells us that the result is 0. If the answer is *no* then we continue on the right branch, knowing now that the number in hand is in the set $\{n \mid n \geqslant 1\}$. The next question asks whether the number *is exactly* 1 or not. If yes, we are done, otherwise we continue as before, until the result is reached.

Figure 2 shows a more interesting game for natural numbers in $\{0..15\}$. This game proceeds by asking whether the number in hand is greater than the *median*

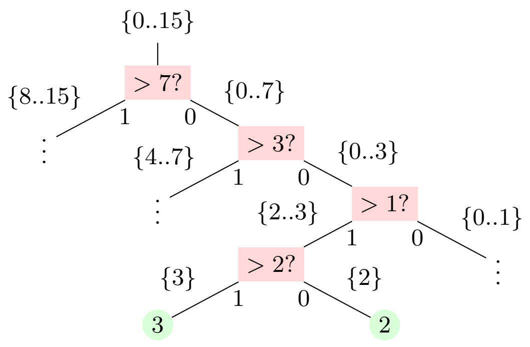


Fig. 2. Binary game for {0..15}

element in the current range. For example, the first question asks of a number $n$ whether $n > 7$, splitting the range into disjoint parts {8..15} and $n \in \{0..7\}$. If $n \in \{8..15\}$ we play the game given by the left subtree. If $n \in \{0..7\}$ we play the game given by the right subtree.

In both games, the encoding of a value can be determined by labelling all left edges with 1 and all right edges with 0, and returning the path from the root to the value. Conversely, to decode, we interpret the input bitstream as a path down the tree. So in the game of Figure 1, a number $n \in \mathbb{N}$ is encoded in unary as $n$ zeroes followed by a one, and in the game of Figure 2, a number $n \in \{0..15\}$ is encoded as 4-bit binary, as expected. For example, the encoding of 2 is 0010 and 3 is 0011. There is one more difference between the two games: the game of Figure 1 is infinite whereas the game of Figure 2 is finite.

It's clear that question-and-answer games give rise to codes that are *unambiguous*: any particular bitstring can be the code for just one value. Moreover, the one-question-at-a-time nature of games ensures that no valid code is the prefix of another valid code. In the literature on coding theory, such an encoding scheme is called *prefix-free* or just a *prefix code* for short (Salomon, 2008).

Notice two properties common to the games of Figure 1 and 2: every value in the domain is represented by some leaf node (we call such games *total*), and each question strictly partitions the domain (we call such games *proper*). Games satisfying both properties give rise to codecs with the following property: any bitstring is a prefix of or has a prefix that is a code for some value. This is the 'every bit counts' property of the title. Note though that if a prefix of a bitstring is a code for some value, the rest of that bitstring is simply junk. In Section 4 we pin these ideas down with theorems.

But how can we actually *compute* with games? We've explained the basic principles in terms of set membership and potentially infinite trees, and we need to translate these ideas into code.

- We must represent *infinite* games without constructing all the leaf nodes ahead-of-time. This is easy: just construct the game tree *lazily*.
- We need something corresponding to 'a set of possible values', which we've

been writing on the arcs in our diagrams. *Types* are the answer here, sometimes with additional implicit invariants; for example, in Haskell, '`Int`s between 4 and 7'.

- We must capture the splitting of the domain into two disjoint parts. This is solved by *type isomorphisms* of the form $\tau \cong \tau_1 + \tau_2$, with $\tau_1$ representing the domain of the left subtree (corresponding to answering *yes* to the question) and $\tau_2$ representing the domain of the right subtree (corresponding to *no*).
- Lastly, we need a means of using this splitting to query the data (when encoding), and to construct the data (when decoding). Type isomorphisms provide a very elegant solution to this task: we simply use the maps associated with the isomorphism.

Let's get concrete with some code!

## 3 Games in Haskell and Coq

We'll dive straight in, with a data type for games, in Haskell:

```
data Game t where
  Single :: ISO t () → Game t
  Split  :: ISO t (Either t1 t2) → Game t1 → Game t2 → Game t
```

A value of type `Game t` represents a game (strictly speaking, a *strategy* for playing a game) for domain `t`. Its leaves are built with `Single` and represent singletons, and its nodes are built with `Split` and represent a splitting of the domain into two parts. The leaves carry a representation of an isomorphism between `t` and the unit type. The nodes carry a representation of an isomorphism between `t` and the discriminated union of `t1` and `t2`, and two subtrees of type `Game t1` and `Game t2`.[2]

The definition in Coq is very similar, declaring `Game` to be a coinductive type in order to support infinite games:

```
CoInductive Game t :=
 | Single : ISO t unit → Game t
 | Split t1 t2 : ISO t (t1 + t2) → Game t1 → Game t2 → Game t.
```

What is `ISO`? It's just a pair of maps witnessing an isomorphism:

```
data ISO t s = Iso (t → s) (s → t)
```

In Coq, the type also records *proofs* of the left inverse and right inverse properties of the maps:

```
Structure ISO t s := Iso { map:> t → s; inv: s → t;
  leftInv: ∀ x, inv (map x) = x; rightInv: ∀ y, map (inv y) = y }.
```

Without further ado we write a *generic* encoder and decoder, once and for all. We use `Bit` for binary digits rather than `Bool` so that output is more readable:

[2] The type variables `t1` and `t2` are *existential variables*, not part of vanilla Haskell 98, but supported by all modern Haskell compilers.

```
data Bit = O | I
```

In Coq, we define a coinductive type for possibly-infinite lists of bits, which plays the role of Haskell's lazy lists:

```
Inductive Bit := O | I.
CoInductive Bits := nilB | consB (b: Bit) (bs: Bits).
```

An encoder for type `t` takes a game of type `Game t` and a value of type `t`, and produces a list of bits of type `Bit`. the Coq code is similar, except that we must make explicit that `enc` may not terminate, by returning a value of type `Bits`:

```
enc :: Game t → t → [Bit]
enc g x =
  case g of
    Single _ → []
    Split (Iso ask _) g1 g2 →
      case ask x of
      Left  y → I : enc g1 y
      Right y → O : enc g2 y
```

```
CoFixpoint enc t (g: Game t) :=
match g with
| Single _⇒ fun x⇒ nilB
| Split _ _ i g1 g2⇒ fun x⇒
  match i x with
  | inl y⇒ consB I (enc g1 y)
  | inr y⇒ consB O (enc g2 y)
  end end.
```

If the game we are playing is a `Single` leaf, then `t` must be a singleton, so we need no bits to encode `t`, and just return the empty list. If the game is a `Split` node, we `ask` how `x` of type `t` can become either a value of type `t1` or `t2`, for some `t1` and `t2` that split type `t` disjointly in two. Depending on the answer we output `I` or `O` and continue playing either the sub-game `g1` or `g2`.

A decoder is also simple to write:

```
dec :: Game t → [Bit]
       → Maybe (t, [Bit])
dec g l = case g of
  Single (Iso _ bld) →
    Just (bld (), l)
  Split (Iso _ bld) g1 g2 →
    case l of
      [] → Nothing
      I : l →
       do (y,r) ← dec g1 l
          Just(bld (Left y), r)
      O : l →
       do (y,r) ← dec g2 l
          Just(bld (Right y), r)
```

```
Fixpoint dec t (g: Game t) l :=
match g with
| Single i⇒ Some(inv i tt, l)
| Split _ _ i g1 g2⇒
  match l with
  | nil⇒ None
  | I :: l ⇒
    if dec g1 l is Some(y, r)
    then Some(inv i (inl _ y), r)
    else None
  | O :: l ⇒
    if dec g2 l is Some(y, r)
    then Some(inv i (inr _ y), r)
    else None
  end end.
```

The decoder accepts a `Game t` and a bitstring of type `[Bit]`. If the input bitstring is too short to decode a value then `dec` returns `Nothing`. Otherwise it returns a decoded value and the suffix of the input list that was not consumed. If the game is `Single`, then `dec` returns the unique value in `t` by applying the inverse map of the isomorphism on the unit value. No bits are consumed, as no questions need answering! If the game is `Split` and the input list is non-empty then `dec` decodes the rest of the bitstring using either sub-game `g1` or `g2`, depending on whether the first bit is `I` or `O`, building a value of type `t` using the inverse map of the isomorphism.

## *3.1 Number games*

These simple definitions already suffice for a range of numeric encodings. We define aliases `Nat` and `Pos` for the Haskell type `Int` to document when our integers are non-negative or positive; in our Coq development these are precise types `nat` and `positive`.

*Unary naturals.* The game of Figure 1 can be expressed as follows:

```
-- ∀ k:nat, Game { x | x ⩾ k }
geNatGame :: Nat → Game Nat
geNatGame k = Split (splitIso (== k))
                    (Single (singleIso k))
                    (geNatGame (k+1))
```

The function `geNatGame` returns a game for natural numbers greater than or equal to its parameter `k`. It consists of a `Split` node whose left subtree is a `Single`ton node for `k`, and whose right subtree is a game for values greater than or equal to `k+1`. The isomorphisms `singleIso` and `splitIso` are used to express singleton values and a partitioning of the set of values respectively. Their signatures and definitions are presented in Figure 3 and 4, along with some other basic isomorphisms that we shall use throughout the paper.

In this game, the isomorphisms just add clutter to the code: one might ask why we didn't define a `Game` type with elements at the leaves and simple predicates in the nodes. But isomorphisms show their true colours when they are used to map between different *representations* or possibly even different *types* of data.

*Unary naturals, revisited.* Consider this alternative game for natural numbers:

```
unitGame :: Game ()
unitGame = Single (Iso id id)

unaryNatGame :: Game Nat
unaryNatGame = Split succIso unitGame unaryNatGame
```

This time we're exploiting the isomorphism $\mathbb{N} \cong \mathbf{1} + \mathbb{N}$, presented in Figure 3. Let's see how it's used in the game. When encoding a natural number $n$, we `ask` whether it's zero or not using the forward map of the isomorphism to get answers of the form `Left ()` or `Right` $(n-1)$, capturing both the *yes*/*no* 'answer' to the question

$\boxed{\{x\} \cong \mathbf{1}}$

```
Definition singleIso a (k:a) : ISO { x | x=k } unit.
```

```
singleIso :: a → ISO a ()
singleIso x = Iso (const ()) (const x)
```

$\boxed{X \cong Y + (X \setminus Y)}$

```
Definition splitIso a (p: a→ bool): ISO a ({y | p y=true} + {y | p y=false}).
```

```
splitIso :: (a → Bool) → ISO a (Either a a)
splitIso p = Iso ask bld
  where ask x = if p x then Left x else Right x
        bld (Left y)  = y
        bld (Right y) = y
```

$\boxed{\mathbb{B} \cong \mathbf{1} + \mathbf{1}}$

```
boolIso : ISO bool (unit + unit).
```

```
boolIso :: ISO Bool (Either () ())
boolIso = Iso ask bld where ask True        = Left ()
                            ask False       = Right ()
                            bld (Left ())   = True
                            bld (Right ())  = False
```

$\boxed{\mathbb{N} \cong \mathbf{1} + \mathbb{N}}$

```
succIso : ISO nat (unit + nat).
```

```
succIso :: ISO Nat (Either () Nat)
succIso = Iso ask bld where ask 0          = Left ()
                            ask n          = Right (n - 1)
                            bld (Left ())  = 0
                            bld (Right n)  = n + 1
```

$\boxed{\mathbb{N} \cong \mathbb{N}^{+}}$

```
natPosIso : ISO nat positive.
```

```
natPosIso :: ISO Nat Pos
natPosIso = Iso succ pred
```

Fig. 3. Some useful isomorphisms (I)

and data with which to continue playing the game. If the answer is `Left ()` then we just play the trivial `unitGame` on the value `()`, otherwise we have `Right` $(n-1)$ and play the very same `unaryNatGame` for the value $n-1$. When decoding, we apply the inverse map of the isomorphism to `build` data with `Left ()` or `Right` $x$ as determined by the next bit in the input stream.

We can test our game using the generic `enc` and `dec` functions:

```
> enc unaryNatGame 3
[O,O,O,I]
> enc unaryNatGame 2
[O,O,I]
```

$\boxed{\mathbb{N} \cong \mathbb{N} + \mathbb{N}}$

```
parityIso : ISO nat (nat + nat).
```

```
parityIso :: ISO Nat (Either Nat Nat)
parityIso = Iso ask bld
  where ask n = if even n then Right (n `div` 2) else Left (n `div` 2)
        bld (Left m)  = m * 2 + 1
        bld (Right m) = m * 2
```

$\boxed{X^\star \cong \mathbf{1} + X \times X^\star}$

```
listIso t : ISO (list t) (unit + t * list t).
```

```
listIso :: ISO [t] (Either () (t,[t]))
listIso = Iso ask bld where ask []       = Left ()
                            ask (x:xs)   = Right (x,xs)
                            bld (Left ())      = []
                            bld (Right (x,xs)) = x:xs
```

$\boxed{X^+ \cong X \times X^\star}$

```
nonemptyIso t : ISO {x:list t | x<>nil} (t * list t).
```

```
nonemptyIso :: ISO [t] (t,[t])
nonemptyIso = Iso ask bld where ask (x:xs) = (x,xs)
                                bld (x,xs) = x:xs
```

$\boxed{X^\star \cong \Sigma n : \mathbb{N}.X^n}$

```
depListIso t: ISO (list t) {n:nat & t^n}.
```

```
depListIso :: ISO [t] (Nat,[t])
depListIso = Iso ask bld where ask xs = (length xs, xs)
                               bld (n,xs) = xs
```

Fig. 4. Some useful isomorphisms (II)

```
> dec unaryNatGame [O,O,I]
Just (2,[])
```

*Finite ranges.* How about the range encoding for natural numbers, sketched in Figure 2? That's easy:

```
-- ∀ m n : nat, Game { x | m ⩽ x ⩽ n }
rangeGame :: Nat → Nat → Game Nat
rangeGame m n | m == n = Single (singleIso m)
rangeGame m n = Split (splitIso (> mid))
                      (rangeGame (mid+1) n)
                      (rangeGame m mid) where mid = (m + n) `div` 2
```

Let's try it out:

```
> enc (rangeGame 0 15) 5
[O,I,O,I]
> dec (rangeGame 0 15) [O,I,O,I]
```

```
Just (5,[])
```

*Binary naturals.* The range encoding results in a logarithmic coding scheme, but only works for naturals in a finite range. Can we give a general logarithmic scheme for arbitrary naturals? Yes, and here is the protocol: we first ask if the number $n$ is zero or not, making use of `succIso` again. If yes, we are done. If not, we ask whether $n-1$ is divisible by 2 or not, making use of `parityIso` from Figure 4 that captures the isomorphism $\mathbb{N} \cong \mathbb{N} + \mathbb{N}$. Here is the code:

```
binNatGame :: Game Nat
binNatGame = Split succIso unitGame $
             Split parityIso binNatGame binNatGame
```

The \$ sign above is just Haskell notation for infix application. We can test this game; for example:

```
> enc binNatGame 8
[O,I,O,O,O,O,I]
> dec binNatGame [O,I,O,O,O,O,I]
Just (8,[])
> enc binNatGame 16
[O,I,O,O,O,O,O,O,I]
```

After staring at the output for a few moments one observes that the encoding takes double the bits (plus one) that one would expect for a logarithmic code. This is because before every step, an extra bit is consumed to check whether the number is zero or not. The final extra `I` terminates the code. In Section 4 we explain how the extra bits result in *prefix codes*, a property that our methodology is designed to validate by construction.

### 3.2 Game combinators

To build games for structured types we provide combinators that construct complex games from simple ones.

*Constant.* Our first combinator is trivial, making use of the isomorphism between the unit type and singletons. It is up to the programmer to ensure that the encoder for `constGame k` is only ever applied to the value `k`.

```
constGame :: t → Game t
constGame k = Single (singleIso k)
```

In Coq, this obligation is expressed in the type of the combinator:

```
Definition constGame t (k:t): Game {x | x=k} := Single (singleIso k).
```

$\boxed{A \cong A}$

```
idI :: ISO a a
```

$\boxed{A \cong B \Rightarrow B \cong A}$

```
invI :: ISO a b → ISO b a
```

$\boxed{A \cong B \wedge B \cong C \Rightarrow A \cong C}$

```
seqI :: ISO a b → ISO b c → ISO a c
```

$\boxed{A \cong B \wedge C \cong D \Rightarrow A \times C \cong B \times D}$

```
prodI :: ISO a b → ISO c d → ISO (a,c) (b,d)
```

$\boxed{A \cong B \wedge C \cong D \Rightarrow A + C \cong B + D}$

```
sumI :: ISO a b → ISO c d → ISO (Either a c) (Either b d)
```

$\boxed{A \times B \cong B \times A}$

```
swapProdI :: ISO (a,b) (b,a)
```

$\boxed{A + B \cong B + A}$

```
swapSumI :: ISO (Either a b) (Either b a)
```

$\boxed{A \times (B \times C) \cong (A \times B) \times C}$

```
assocProdI :: ISO (a,(b,c)) ((a,b),c)
```

$\boxed{A + (B + C) \cong (A + B) + C}$

```
assocSumI :: ISO (Either a (Either b c)) (Either (Either a b) c)
```

$\boxed{\mathbf{1} \times A \cong A}$

```
prodLUnitI :: ISO ((),a) a
```

$\boxed{A \times \mathbf{1} \cong A}$

```
prodRUnitI :: ISO (a,()) a
```

$\boxed{A \times (B + C) \cong (A \times B) + (A \times C)}$

```
prodRSumI :: ISO (a,Either b c) (Either (a,b) (a,c))
```

$\boxed{(B + C) \times A \cong (B \times A) + (C \times A)}$

```
prodLSumI :: ISO (Either b c,a) (Either (b,a) (c,a))
```

Fig. 5. Isomorphism combinator signatures

*Cast.* The combinator (+>) transforms a game for t into a game for s, given that s is isomorphic to t.

```
(+>) :: Game t → ISO s t → Game s
(Single j) +> i      = Single (i ‘seqI‘ j)
(Split j g1 g2) +> i = Split  (i ‘seqI‘ j) g1 g2
```

What is seqI? It is a combinator *on isomorphisms*, which wires two isomorphisms together. In fact, combining isomorphisms together in many ways is generally useful, so we define a small library of isomorphism combinators. Their signatures are given in Figure 5 and their implementation (and proof) is entirely straightforward.

*Choice.* It's dead easy to construct a game for the sum of two types, if we are given games for each. The `sumGame` combinator is so simple that it hardly has a reason to exist as a separate definition:

```
sumGame :: Game t → Game s → Game (Either t s)
sumGame = Split idI
```

*Composition.* Suppose we are given a game $g_1$ of type `Game t` and a game $g_2$ of type `Game s`. How can we build a game for the product `(t,s)`? A simple strategy is to play $g_1$, the game for `t`, and at the leaves play $g_2$, the game for `s`. The `prodGame` combinator achieves this, as follows:

```
prodGame :: Game t → Game s → Game (t,s)
prodGame (Single iso) g2 = g2 +> prodI iso idI ‘seqI‘ prodLUnitI
prodGame (Split iso g1a g1b) g2 =
  Split (prodI iso idI ‘seqI‘ prodLSumI)
        (prodGame g1a g2)
        (prodGame g1b g2)
```

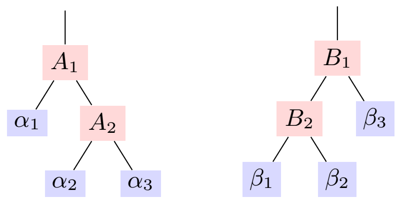
If the game for `t` is a singleton node, then we play `g2`, which is the game for `s`. However, that will return a `Game s`, whereas we'd like a `Game (t,s)`. But from the type of the `Single` constructor we know that `t` is the unit type `()`, and so we *coerce* `g2` to the appropriate type using combinators from Figure 5 to construct an isomorphism between `s` and `((),s)`. In the case of a `Split` node, we are given an isomorphism `iso` of type `ISO t (Either t1 t2)` for unknown types `t1` and `t2`, and we create a new `Split` node whose subtrees are constructed recursively, and whose isomorphism of type `ISO (t,s) (Either (t1,s) (t2,s))` is again constructed using the combinators from Figure 5.

*Lists.* What can we do with `prodGame`? We can build more complex combinators, such as the following recursive `listGame` that encodes lists:

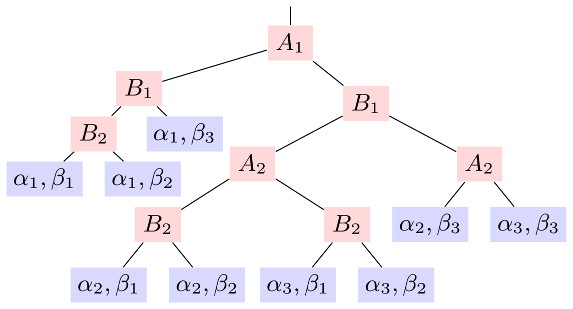
```
listGame :: Game t → Game [t]
listGame g = Split listIso unitGame (prodGame g (listGame g))
```

It takes a game for `t` and produces a game for lists of `t`. The question asked by `listIso` is whether the list is empty or not. If empty then we play the left sub-game – a singleton node – and if non-empty then we play the right sub-game, consisting of a game for the head of the list followed by the list game for the tail of the list. This is just the product `prodGame g (listGame g)`.

*Composition by interleaving.* Recall that `prodGame` pastes copies of the second game in the leaves of the first game. However if the first component of a pair is an infinite stream, and we'd like an online decoder, then `prodGame` is useless since it requires the first value to be decoded in its entirety before we can go on in decoding the second component. An alternative approach is to *interleave* the bits of the two games. We illustrate this graphically, starting with example games given below:



Interleaving the two games, starting with the left-hand game gives:



The `ilGame` below does that by playing a bit from the game on the left, but always 'flipping' the order of the games in the recursive calls. Its definition is similar to `prodGame`, with isomorphism plumbing adjusted appropriately:

```
ilGame :: Game t → Game s → Game (t,s)
ilGame (Single iso) g2 = g2 +> prodI iso idI ‘seqI‘ prodLUnitI
ilGame (Split iso g1a g1b) g2 =
  Split (swapProdI ‘seqI‘ prodI idI iso ‘seqI‘ prodRSumI)
        (ilGame g2 g1a)
        (ilGame g2 g1b)
```

The resulting encoding of product values of course differs between `ilGame` and `prodGame`, although it will use exactly the same number of bits.

*Dependent composition.* Suppose that, after having decoded a value `x` of type `t`, we wish to play a game whose strategy *depends* on `x`. For example, given a game for natural numbers, and a game for lists of a particular size, we could create a game for arbitrary lists paired up with their size. We can do this with the help of a *dependent composition* game combinator.

```
depGame :: Game t → (t → Game s) → Game (t,s)
depGame (Single (iso@(Iso _ inv))) f =
    f (inv ()) +> prodI iso idI ‘seqI‘ prodLUnitI
depGame (Split (iso@(Iso _ inv)) g1a g1b) f
  = Split (prodI iso idI ‘seqI‘ prodLSumI)
          (depGame g1a (f ∘ inv ∘ Left))
          (depGame g1b (f ∘ inv ∘ Right))
```

The definition of `depGame` resembles the definition of `prodGame`, but notice how in the `Single` case we apply the `f` function to the singleton value to determine the game we must play next.

The type of the `depGame` combinator is especially illuminating in Coq:

```
depGame: ∀ t s, Game t → (∀ x:t, Game (s x)) → Game { x:t & s x }
```

Here, the second game has a dependent function type (a Π type) whose result is a game whose *type* can depend on the value of the argument; the resulting game is for a dependent pair type (a Σ type) the type of whose second component depends on the value of the first component.

As might be expected, the original `prodGame` can be expressed very easily in terms of the more general `depGame`:

```
prodGameAlt :: Game t → Game s → Game (t,s)
prodGameAlt g1 g2 = depGame g1 (const g2)
```

Finally, notice that the Haskell type of `depGame` looks similar to the type of monadic bind – it is a very interesting future work to explore the monadic structure of games.

*Lists, revisited.* We can use `depGame` to create an alternative encoding for lists. Suppose we are given a function `vecGame` that builds a game for lists of a given length:

```
vecGame :: Game t → Nat → Game [t]
vecGame g 0 = constGame []
vecGame g n = prodGame g (vecGame g (n - 1)) +> nonemptyIso
```

We can then define a game for lists paired with their length, and use the isomorphism `depListIso` from Figure 4 to derive a new game for lists, as follows:

```
listGame' :: Game Nat → Game t → Game [t]
listGame' natGame g = depGame natGame (vecGame g) +> depListIso
```

The game is parameterized on a `Game Nat` used to encode the length. It's interesting to observe that `listGame' unaryNatGame` will use exactly the same number of bits as our original `listGame`: in effect, the latter encodes the length of the list as a unary representation interleaved with the elements of the list.

*Numbers, revisited.* Taking inspiration from this alternative encoding of lists, we now consider encoding a (positive) number by taking its binary representation, dropping the most-significant bit, and preceding it by some representation of the number of bits.

This family of codes (Salomon, 2008) was devised by Elias (1975) and can be implemented straightforwardly as follows:

```
eliasGame :: Game Nat → Game Pos
eliasGame natGame = depGame natGame binGame +> binIso
  where binGame 0 = constGame 1
        binGame n = Split parityIso (binGame (n-1)) (binGame (n-1))
```

```
    binIso = Iso (λp → (log2 p, p)) snd
    log2 1 = 0
    log2 p = 1 + log2 (p `div` 2)
```

Let $\mathrm{BIN}_n$ be the set $\{x \in \mathbb{N} \mid 2^n \leqslant x < 2^{n+1}\}$, in other words, those positive numbers whose standard binary encoding contains $n+1$ bits. Then `binGame` $n$ is a game for $\mathrm{BIN}_n$, which cunningly re-uses `parityIso` as the isomorphism $\mathrm{BIN}_{n+1} \cong \mathrm{BIN}_n + \mathrm{BIN}_n$. (Note that, in contrast to the original presentation by Elias, this representation is little-endian, or least significant bit first.) The `binIso` code expresses the isomorphism between $\mathbb{N}^+$ (the positive natural numbers) and $\{\mathrm{BIN}_n\}_{n\in\mathbb{N}}$ (the family of all $n$-bit binary representations).

We can instantiate `natGame` with `unaryNatGame` to get the $\gamma$ code described by Elias[3]:

```
gammaGame :: Game Pos
gammaGame = eliasGame unaryNatGame
```

To represent a number we first encode in unary the number of bits required for its binary representation, followed by the bits (least significant first) of the binary representation, dropping the most significant bit. So, for example, the number 34 is written in six bits as 100010, dropping the most significant bit and writing it in little-endian form produces 01000, and so its code is 000001 01000. Observe that `gammaGame` uses the same number of bits as our `binNatGame` from Section 3.1: in effect, the latter encodes the length of the binary as a unary representation interleaved with the bits of the binary.

The clever bit is that we can now bootstrap. First observe that we can use the $\gamma$ code defined above to obtain the $\delta$ code of Elias:

```
deltaGame :: Game Pos
deltaGame = eliasGame (gammaGame +> natPosIso)
```

For example, the number 34 would be represented as 001 01 01000. Why? Because 34 in little-endian binary without msb is 010000, which has five bits, and we now use the $\gamma$ code – after applying the isomorphism $\mathbb{N} \cong \mathbb{N}^+$ in order to use a game that expects positive numbers – to encode $5 \in \mathbb{N}$ as 001 01.

Finally, we can apply the power of recursion to implement the limiting case of bootstrapping – called the $\omega$ code by Elias for obvious reasons – in a single line:

```
omegaGame :: Game Pos
omegaGame =
  Split (splitIso (== 1)) (constGame 1) (eliasGame omegaGame)
```

The $\omega$ encoding of 34 is 0001 0 10 01000. The last 5 bits 01000 are the little-endian, msb-dropped representation of 34. The next 2 bits 10 are the little-endian, msb-dropped representation of 5. The next 1 bit 0 is the little-endian, msb-dropped representation of 2. Finally, the first four bits is a unary encoding of the number of subsequent groups of bits.

[3] Elias used $\alpha$ for unary, $\beta$ for standard binary, and continued with $\gamma$ for this code

| symbol | (a) fixed length | (b) variable length | (c) uniquely decodable | (d) prefix-free | (e) redundant | (f) complete |
|---|---|---|---|---|---|---|
| A | 00 | 0 | 0 | 0 | 0 | 0 |
| B | 01 | 10 | 01 | 100 | 100 | 100 |
| C | 10 | 101 | 011 | 101 | 101 | 101 |
| D | 11 | 111 | 0111 | 111 | 111/110 | 11 |

Fig. 6. Code zoo

To understand this, first observe that `eliasGame` can be assigned the more refined type `Game Pos →Game {x | x ⩾2}`.

Complete this!!!

## 4 Properties of games

We now turn to the formal properties of game-based codecs. The basic correctness theorem follows simply from the validity of isomorphisms present in the games, and termination and the *every bit counts* property of the title follow from some easily-stated requirements on game trees.

We present the statements of all theorems in this section in Coq. The results apply to Haskell code, with a couple of provisos. First, results concerning precise Coq types such as `nat` or `{ x | x > 5 }` apply only to the appropriate subdomain of the less precise Haskell type. For example, all bets are off when feeding a negative value of Haskell type `Int` to `enc unaryNatGame`. Furthermore, nothing is said about the encoding of infinite values, such as `enc (listGame natGame) ones` where `ones` is the infinite stream of 1's. Note, though, that we do model infinite games in Coq (through a `CoInductive` type), and also the possibly non-terminating behaviour of `enc` (through the type `Bits` of finite and infinite lists of bits). In order to capture termination of the encoder, we write `enc g x = fromList l`, where l is of type `list Bit`, and `fromList` embeds finite lists into the `Bits` type, as follows:

```
Definition fromList := fold_right consB nilB.
```

### *4.1 Correctness*

We're interested only in lossless codes, so at the very least we expect a precise round-trip property: encoding followed by decoding should return us to where we started. In fact, for a correctly-constructed game we can prove the following more general theorem, which asserts that if `x` encodes to a finite bitstring `l`, then the decoding of any extension `e` of `l` returns `x` together with the extension.

```
Theorem Roundtrip: ∀ l t (g: Game t) x,
   enc g x = fromList l→ ∀ e, dec g (l ++ e) = Some (x, e).
```

The proof is by induction on `l` and makes use of the `leftInv` property from the isomorphisms embedded in the games.

This theorem packages up several facts about the codecs induced by games. The first of these is simply that the encoding function is injective: no two values are assigned the same code.

```
Corollary Injectivity: ∀ l t (g: Game t) x y,
  (enc g x = fromList l  ∧  enc g y = fromList l) → x = y.
```

Clearly injectivity is a necessary property of an encoding function, but it's not sufficient. Consider the zoo of codecs for a four-element type shown in Figure 6. The first of these is a simple two-bit fixed-length code. The second is a more interesting *variable-length* code. As a self-contained code, it satisfies the basic requirement of injectivity. But if the code is extended to *sequences* of symbols simply by appending their codes, then it becomes ambiguous. For example, consider the code 1010: we cannot tell whether it represents the sequence $CA$ or the sequence $BB$. In the literature on coding theory, a variable-length code for symbols is said to be *uniquely decodable* (UD) if its extension to sequences is injective (Salomon, 2008; MacKay, 2003).

A second corollary of the round-trip theorem is that codecs induced by games are uniquely decodable. Here `t^n` is an `n`-fold product of `t`, and `encvec` is the `n`-fold appending of `enc` applied to the elements of `t^n`.

```
Corollary UD: ∀ t n (g: Game t) l (v w: t^n),
  (encvec g v = fromList l  ∧  encvec g w = fromList l) → v = w.
```

Now consider codec (c) in Figure 6. It is uniquely decodable, as the initial zero in each code acts as a kind of 'punctuation'. However, it's necessary for the decoder to 'look ahead' in order to determine the end-point of each symbol's code. The one-question-at-a-time nature of our games prevents such look-ahead, and thus ensures that codes are *prefix-free* (or a *prefix code* for short), meaning that no prefix of a valid code can itself be a valid code. It is easy to see that codec (d) in Figure 6 has this property. For prefix codes, we can stop decoding at the first successfully decoded value: no look-ahead is required. This is a very useful property, as it enables easy *composition* of codecs, which we have seen already in combinators such as `prodGame` and `listGame`.

```
Corollary Prefix: ∀ l e t (g: Game t) x y,
  (enc g x = fromList l  ∧  enc g y = fromList (l ++ e)) → x = y.
```

An important result from the theory of codes states that for any UD code there exists a prefix code with the same code lengths. Hence we are not losing out by restricting ourselves to prefix codes.

It is worth pausing for a moment to return briefly to the game `binNatGame` from Section 3. Observe that the 'standard' binary encoding for natural numbers *is not* a prefix code (it's not even uniquely decodable). For example the encoding of 3 is 11 and the encoding of 7 is 111. The extra bits inserted by `binNatGame` are necessary to convert the standard encoding to one which *is* a prefix encoding. The anticipated

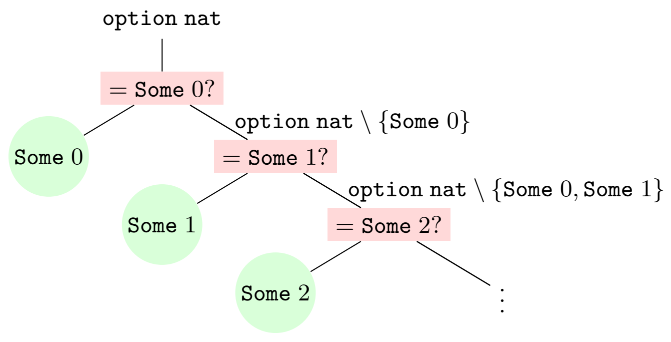


Fig. 7. Game for optional naturals

downside are the inserted 'punctuation' bits that double the size of the encoding (but keep it $\Theta(\log n)$).

## *4.2 Termination*

Thus far our main theorem and its corollaries describe the results of *terminating* encoders. Although in traditional coding theory termination of encoding for any value is taken for granted, it doesn't follow automatically for our game-based codecs.

Figure 7 presents a somewhat odd game for the type `option nat`. At every step $i$, the game asks whether the value in hand is `Some` $i$, or any other value in the type `option nat`. Notice that when asked to encode a value `None` the encoder will simply play the game for ever, diverging.

That's certainly no good! Fortunately, we can require games to be *total*, meaning that every element in the domain is represented by some leaf node. Given a game $g$ of type `Game t` and value $x$ of type `t`, we write $g \rightsquigarrow x$, read '$g$ generates $x$', and defined inductively as follows:

$$\frac{}{\texttt{Single}\ i \rightsquigarrow \texttt{inv}\ i\ \texttt{tt}} \qquad \frac{g_1 \rightsquigarrow x_1}{\texttt{Split}\ i\ g_1\ g_2 \rightsquigarrow \texttt{inv}\ i\ (\texttt{inl}\ x_1)} \qquad \frac{g_2 \rightsquigarrow x_2}{\texttt{Split}\ i\ g_1\ g_2 \rightsquigarrow \texttt{inv}\ i\ (\texttt{inr}\ x_2)}$$

The definition of total game is then easy:

```
Definition Total t (g: Game t) := ∀ x, g ⇝ x.
```

The reader can check that, with the exception of the game in Figure 7, the games presented so far are total; furthermore the combinators on games preserve totality.

We can then prove that if a game is total then `enc` terminates on all inputs.

```
Theorem Termination: ∀ t (g: Game t),
  Total g ↔ ∀ x, ∃ l, enc g x = fromList l.
```

The proof uses an auxiliary lemma which states that for any $x$, if $g \rightsquigarrow x$ then `enc` $g$ $x$ terminates. The proof proceeds by induction on the structure of the derivation of $g \rightsquigarrow x$.

### *4.3 Redundancy*

Now consider codec (e) in Figure 6, in which symbol $D$ is assigned *two* codes, 110 and 111. The third bit of this code is wasted, as the first two bits uniquely determine the value. Of course the encoding function `enc` induced by a game must produce just one code, but can the decoding function `dec` accept more than one code for a single value? Fortunately, construction of games from type isomorphisms guarantees not only that two values will never be assigned the same code, but also that two codes cannot represent the same value. We show this by first proving a 'reverse' round-trip property:

```
Theorem ReverseRoundtrip: ∀ l t (g: Game t) x s,
  dec g l = Some (x, s) → ∃ p, enc g x = fromList p ∧ p ++ s = l.
```

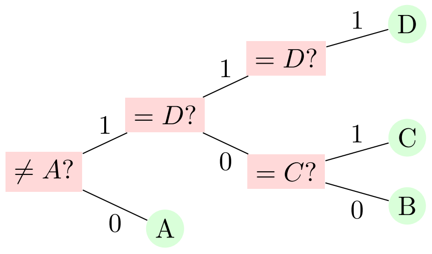
The proof is by induction on the length of `l`, making use of the `rightInv` property from the isomorphisms embedded in the games.

Injectivity of decoding is a simple corollary.

```
Corollary DecInjectivity: ∀ t (g: Game t) l l' x,
  (dec g l = Some (x, nil) ∧ dec g l' = Some (x, nil)) → l = l'.
```

### *4.4 Every bit counts*

Study once more the prefix code (d) in Figure 6. As with codec (e), it's clear that the final bit in the code 111 for $D$ is redundant, and can be interpreted as asking the same question twice:



We can implement this codec in Haskell as follows:

```
data Sym = A | B | C | D deriving (Eq, Show)

voidGame :: Game t -- precondition: t is uninhabited
voidGame = Split (splitIso (const True)) voidGame voidGame

badSymGame :: Game Sym
badSymGame =
  Split (splitIso (/= A))
    (Split (splitIso (== D))
      (Split (splitIso (== D)) (constGame D) voidGame)
      (Split (splitIso (== C)) (constGame C) (constGame B)))
    (constGame A)
```

It may take a little head-scratching to work out what's going on! The first time that `(== D)` is encountered, the game partitions the possible values into {`D`} and {`B`,`C`}. But it then asks the same 'question' in the left-hand branch, even though we're now in a singleton set, so this time partitioning the values into {`D`} and {}. The right-hand branch is dead *i.e.* we have a domain that is not inhabited – hence the use of `voidGame` in the code.

Here's a session that illustrates the `badSymGame` behaviour:

```
> enc badSymGame D
[I,I,I]
> dec badSymGame [I,I,I]
Just (D,[])
> dec badSymGame [I,I,O]
Nothing
```

For domains more complex than `Sym`, such 'stupid questions' are harder to spot. Suppose, for example, that in the game for programs described in the introduction, the first question had been '*Are you a variable?*' Because we know that the program under inspection is closed, this question is silly, and we already know that the answer is *no*.

We call a game *proper* if every isomorphism in `Split` nodes is a proper splitting of the domain; or, equivalently, if for every subgame in the game tree, its type is inhabited. It is immediate that `voidGame` is not a proper game and consequently `badSymGame` is not proper either.

For proper games we can show that decoding *only* fails if the input is simply incomplete, *i.e.* it has some valid extension.

```
Theorem ProperFailure: ∀ t (g: Game t), (Proper g ∧ Total g)→
  ∀ l, dec g l = None→∃ e, ∃ x, enc g x = fromList (l ++ e).
```

In literature on coding theory, a UD code is said to be *complete* if adding any code word to the code table results in a non-UD codec. Equivalently, every finite bitstring is either a prefix of a valid code or has a prefix which is a valid code. The final codec (f) in Figure 6 is complete, as are many coding schemes such as the well-known Huffman coding. Completeness is a straightforward corollary of the `ReverseRoundtrip` and `ProperFailure` theorems.

```
Corollary Completeness: ∀ t (g: Game t), (Proper g ∧ Total g)→
  ∀ l, ∃ x, ∃ l', enc g x = fromList l' ∧ (l' ⊑ l ∨ l ⊑ l').
```

A codec that is complete justifies the slogan *every bit counts*: up to prefix-closure, every codeword is 'used'; every bit read by the decoder is relevant.

## *4.5 On infinite games*

The careful reader will have observed that the `Completeness` theorem requires not only that the game be proper, but also *total*. Consider the following variation of the infinite `binNatGame` from Section 3.1.

```
CoFixpoint badNatGame: Game nat :=
  Split parityIso badNatGame badNatGame.
```

The question asked splits the input set of all natural numbers into two disjoint and inhabited sets: the even and the odd ones. However, there are no singleton nodes in `badNatGame` and hence `Completeness` cannot hold for this game.

As a final observation, notice that even in a total and proper game with infinitely many leaves (such as the natural numbers game in Figure 1) there will be an infinite number of bit strings on which the decoder fails. By König's lemma, in such a game there must exist at least one infinite path, and the decoder will fail on all prefixes of that path.

### *4.6 Summary*

Here is what we have learned in this section.

- Games give rise to uniquely decodable, prefix-free, and non-redundant codes, which round-trip in both directions.
- If the game is total then the encoder terminates on all inputs.
- If additionally the game is proper then every bitstring encodes some value or is the prefix of such a bitstring.

For the the rest of this paper we embark in giving more ambitious and amusing concrete games for sets, several compression schemes, and $\lambda$-terms.

## 5 Sets and maps

So far we have considered primitive and structured data types such as natural numbers, lists and trees, for which games can be constructed in a *type-directed* fashion. Indeed, we could even use *generic programming* techniques (Gibbons, 2007; Hinze *et al.*, 2006) to generate games (and thereby codecs) automatically for such types. (The advanced number games, on the other hand, required some ingenuity.)

But what about other structures such as *sets*, *multisets* or *maps*, in which implicit invariants or equivalences hold, and which our games could be made aware of? For example, consider encoding sets of natural numbers using lists. We know (a) that duplicate elements do not occur, and (b) that the order doesn't matter. We could use `listGame binNatGame` for this type. It would satisfy the basic round-tripping property; however, bits would be 'wasted' in assigning distinct codes to equivalent values such as `[1,2]` and `[2,1]`, and in assigning codes to non-values such as `[1,1]`.

In this section we show how to encode sets, multisets and maps efficiently. First (§5.1), we consider the specific case of sets and multisets of natural numbers, for which it's possible to hand-craft 'delta' encodings in which every bit counts. Next (§5.2), we show how for arbitrary types we can use an ordering on values induced by the game for the type to construct games for sets and multisets of elements of that type, and for finite maps whose domain is that type. Finally (§5.3) we consider the encoding of *permutations* and make an interesting connection with so-called *parsimonious* sorting algorithms.

### *5.1 Hand-crafted games*

What's a good code for the multiset $\{3, 6, 5, 6\}$? We might start by ordering the values to obtain $[3, 5, 6, 6]$ (the Haskell multiset library provides a function `toAscList` that does just this), and then encode this 'canonical representation' using the standard game `listGame binNatGame`. But wait! When encoding the second element, we are wasting the codes for values 0, 1, and 2, as none of them can possibly follow 3 in the ordering. So instead of encoding the value 5 for the second element of the ordered list, we encode 2, the *difference* between the first two elements. Doing the same thing for the other elements, we obtain the list $[3, 2, 1, 0]$, which we can encode using `listGame binNatGame` without wasting any bits. To decode, we reverse the process by adding the differences.

The same idea can be applied to sets, except that the delta is smaller by one, taking account of the fact that the difference between successive elements is never zero.

In Haskell, we implement `diff` and `undiff` functions that respectively compute and apply difference lists.

```
diff :: (Nat → Nat → Nat) → [Nat] → [Nat]
diff minus [] = []
diff minus (x:xs) = x : diff’ x xs
  where diff’ base [] = []
        diff’ base (x:xs) = minus x base : diff’ x xs

undiff :: (Nat → Nat → Nat) → [Nat] → [Nat]
undiff plus [] = []
undiff plus (x:xs) = x : undiff’ x xs
  where undiff’ base [] = []
        undiff’ base (x:xs) = base’ : undiff’ base’ xs
                              where base’ = plus base x
```

The functions are parameterized on subtraction and addition operations, and are instantiated with appropriate concrete operations to obtain games for finite multisets and sets of natural numbers, as follows:

```
natMultisetGame :: Game Nat → Game (MS.MultiSet Nat)
natMultisetGame g = listGame g +> Iso (diff (-) ∘ MS.toAscList)
                                      (MS.fromList ∘ undiff (+))

natSetGame :: Game Nat → Game (Set Nat)
natSetGame g = listGame g +> Iso (diff (λ x y → x-y-1) ∘ toAscList)
                                 (fromList ∘ undiff (λ x y → x+y+1))
```

Here is the set game in action, using our binary encoding of natural numbers on the set $\{3, 6, 5\}$.

```
> enc (listGame binNatGame) [3,6,5]
[O,O,O,O,O,I,O,O,I,O,I,I,O,O,O,O,I,I,I]
```

```
> let l = enc (natSetGame binNatGame) (fromList [3,6,5])
> l
[O,O,O,O,O,I,O,O,O,I,O,I,I]
> dec (natSetGame binNatGame) l
Just (fromList [3,5,6],[])
```

As expected, the encoding is more compact than a vanilla list representation.

## *5.2 Generic games*

What if we want to encode sets of pairs, or sets of sets, or sets of λ-terms? First of all, we need an ordering on elements to derive a canonical list representation for the set. Conveniently, the game for the element type itself gives rise to natural comparison and sorting functions:

```
compareByGame :: Game a → (a → a → Ordering)
compareByGame (Single _) x y = EQ
compareByGame (Split (Iso ask bld) g1 g2) x y =
  case (ask x, ask y) of
    (Left x1 , Left y1)  → compareByGame g1 x1 y1
    (Right x2, Right y2) → compareByGame g2 x2 y2
    (Left x1,  Right y2) → LT
    (Right x2, Left y1)  → GT
sortByGame :: Game a → [a] → [a]
sortByGame g = sortBy (compareByGame g)
```

We can then use the list game on a sorted list, but at each successive element *adapt* the element game so that ‘impossible’ elements are excluded. To do this, we write a function `removeLE` that removes from a game all elements smaller than or equal to a particular element, with respect to the ordering induced by the game. If the resulting game would be empty, then the function returns `Nothing`.

```
removeLE :: Game a → a → Maybe (Game a)
removeLE (Single _) x = Nothing
removeLE (Split (Iso ask bld) g1 g2) x =
  case ask x of
    Left x1 → Just $ case removeLE g1 x1 of
                       Nothing  → g2 +> rightI
                       Just g1' → Split (Iso ask bld) g1' g2
    Right x2 → case removeLE g2 x2 of
      Nothing  → Nothing
      Just g2' → Just (g2' +> rightI)
  where rightI = Iso (getRight ∘ ask) (bld ∘ Right)
```

The code for `listGame` can then be adapted to handle sets:

```
setGame :: Ord a ⇒ Game a → Game (Set a)
setGame g = setGame' g +> Iso (sortByGame g ∘ toList) fromList
```

```
  where setGame’ g = Split listIso unitGame $
                     depGame g $ λx →
                     case removeLE g x of
                       Just g’ → setGame’ g’
                       Nothing → constGame []
```

Notice the dependent composition which takes the value `x` played by game `g`, removes all values no bigger than it from the game, and then recurses.

It’s straightforward to implement a function `removeLT` that removes from a game all items strictly smaller than some value, and then use this to implement a game for multisets. Also easy is a game for finite maps implemented as association lists, since the indices form a set:

```
mapGame :: Game k → Game d → Game [(k,d)]
mapGame gk gd = mapGame’ gk +>
  Iso (sortBy (λ(k1,_) (k2,_) → compareByGame gk k1 k2)) id
    where mapGame’ gk = Split listIso unitGame $
                        depGame (prodGame gk gd) $ λ(k,d) →
                        case removeLE gk k of
                          Just gk’ → mapGame’ gk’
                          Nothing → constGame []
```

### 5.3 Permutations and sorting

We now turn to another example which reveals a connection between complete codes and the notion of *parsimonious* algorithms: a codec for permutations. Formally, a permutation $p$ on elements $1..n$ is just a bijective mapping on integers $1..n$. As an example, the mapping $1 \mapsto 2, 2 \mapsto 1, 3 \mapsto 3$ is such a permutation.

How would we go about representing and encoding/decoding a permutation? We could start by representing a permutation in the standard way, as a list of $n$ *distinct* integers in the range $1..n$. For the permutation above this list is simply $[2, 1, 3]$. Following the pattern applied in previous sections to sets, multisets, and finite maps, we could then use `rangeGame 1 n` for the first element of the list, and then remove elements from this game as the list is traversed.

But there is an interesting alternative: Let us still represent the permutation as a list of distinct elements as before. We may now try to *sort* this list using a comparison-based sorting algorithm, and encode a *trace* of the results from each comparison-test as a sequence of bits. For any permutation, the final sorted list is, of course, `[1..n]`, which is not interesting. But, for a given sorting algorithm the trace of the algorithm could provide a *code* for the permutation.

At first glance this looks rather hairy to implement. But in fact, using our isomorphism-based games it is remarkably easy. Crucially, we do not implement the sorting algorithm for a concrete type of functions on arrays, or lists, or whatever, but instead *parameterize* it both on a type `c` of computations, and on the operations of comparing two elements, transposing two elements, and completing the sort. The comparison and transposition operations can be packaged conveniently in a type class:

```
class Permutator c where
  ifLt :: Nat → Nat → c → c → c
  swapThen :: Nat → Nat → c → c
```

The intention of the `iflt` operation is that `iflt i j c1 c2` compares the elements at indices `i` and `j`, and then takes branch `c1` if the element at `i` is smaller than the element at `j`, otherwise it takes branch `c2`. The `swapThen i j c` operation exchanges the elements at indices `i` and `j`, and then proceeds with computation `c`.

Here is bubble sort expressed using these primitives:

```
bubble :: Permutator c ⇒ Nat → c → c
bubble n finish = bub False 0 (n-1) where
    bub swapped k m =
      if k==m
      then (if swapped then bub False 0 (m-1) else finish)
      else ifLt k (k+1)
             (bub swapped (k+1) m)
             (swapThen k (k+1) (bub True (k+1) m))
```

In addition to the implicit parameterization on `iflt` and `swapThen`, the computation `finish` is executed once the sort algorithm has completed. It is assumed that elements from indices `0` to `n-1` are to be sorted.

Although our purpose is to write codecs for permutations, we can use the sorting algorithm for (obviously) sorting, by declaring an instance of `Permutator` for the type `[a] → [a]`, as follows:

```
swap :: Nat → Nat → [a] → [a]
swap i j p = [ if k==i then p !! j else
               if k==j then p !! i else
               p !! k | k ← [0..length p-1]]

instance Ord a ⇒ Permutator ([a] → [a]) where
  ifLt i j l r s = if s!!i < s!!j then l s else r s
  swapThen i j f = f ∘ swap i j

bubbleSort :: Ord a ⇒ [a] → [a]
bubbleSort s = bubble (length s) id s
```

Alternatively, we can define an instance of `Permutator` for *games* on permutations. The idea is simple. At any point during the game, we will be 'asking questions' of a permutation $p$ that is drawn from a set of possible permutations $P$. Permutations (of length $n$) are represented as lists of length `n` with an extra invariant that their contents are distinct elements in the range `[1..n]`:

```
type Perm = [Nat]
```

Now observe that given distinct indices $i$ and $j$, the isomorphism $P \cong \{p \in P \mid p(i) < p(j)\} + \{p \in P \mid p(i) > p(j)\}$ holds (notice the strict inequalities in both sides due to the fact that all elements in the range of the permutation are distinct), partitioning permutations into those that preserve the 'order' of the $i$'th and $j$'th elements, and those that invert the order. In Haskell, we write:

```
compareIso :: Nat → Nat → ISO Perm (Either Perm Perm)
compareIso i j = splitIso (λp → p!!i < p!!j)
```

Furthermore, observe that $P \cong \{(i \leftrightarrow j) \circ p \mid p \in P\}$, where $i \leftrightarrow j$ denotes the permutation consisting of the transposition of elements $i$ and $j$. In Haskell:

```
swapIso :: Nat → Nat → ISO Perm Perm
swapIso i j = Iso (swap i j) (swap i j)
```

Now we can declare `Game Perm` to be an instance of `Permutator`, with the branching combinator `iflt` implemented by `Split`ting through the `compareIso` isomorphism, and `swapThen` implemented by coercing a game through the `swapIso` isomorphism.

```
instance Permutator (Game Perm) where
  ifLt i j = Split (compareIso i j)
  swapThen i j g = g +> swapIso i j
```

The final piece in the jigsaw is the `finish` parameter to `bubble`, which we instantiate with a singleton game containing the identity permutation:

```
bubbleGame :: Nat → Game Perm
bubbleGame n = bubble n (constGame [1..n])
```

And now we can test it!

```
> enc (bubbleGame 4) [1,2,3,4]
[I,I,I]
> enc (bubbleGame 4) [1,3,2,4]
[I,O,I,I,I]
> dec (bubbleGame 4) [I,O,I,I,I]
Just ([1,3,2,4],[])
```

Observe how the code for the identity permutation has just three bits, because bubble sort traverses the list only once, reporting the result of three comparisons; on the other hand the code for the permutation $[1,3,2,4]$ has five bits, because bubble sort traverses the list twice, reporting the results of three comparisons on the first pass and two on the second. Of course it's possible to use different in-place, comparison-based sorting algorithms to induce permutation codecs whose codes are distributed differently; we've done this for Quicksort, which is included in the code available online. As a side note, in Quicksort not all compare operations are followed by a potential swap, which is the reason for the separation of the two operations in our `Permutator` class. For the purposes of bubble sort we could get away with just a single compare-and-swap primitive.

Correctness of the `compareIso` and `swapIso` isomorphisms implies correctness of the codec. But what about the *completeness*, or ‘every bit counts’ property that we formalized in Section 4? For this to hold, every instance of the `compareIso` isomorphism must be a proper partitioning of the set of permutations. Rather beautifully, this corresponds to a property of sorting algorithms that Knuth and others call *parsimony*: a parsimonious sorting algorithm invokes the comparison operation only on elements whose order it cannot determine from previous comparisons. Most sorting algorithms are parsimonious; a terrible implementation of bubble sort that uniformly performs $n$ traversals is not parsimonious. In Knuth’s words, a parsimonious algorithm “asks no stupid questions” (§15, Knuth, 1992).

So we have the following connection: given a sorting algorithm `sort` implemented parametrically as above, if `sort` is parsimonious then `sort n (constGame [1..n])` implements a complete codec for permutations of size `n`.

## 6 Codes for programs

We’re now ready to return to the problem posed in the introduction: how to construct games for *programs*. As with the types described in the previous section, the challenge is to devise games that satisfy the every-bit-counts property, so that any string of bits represents a unique well-typed program, or is the prefix of such a code.

### *6.1 No types*

First let’s play a game for the untyped $\lambda$-calculus, declared as a Haskell datatype using de Bruijn indexing for variables:

```
data Exp = Var Nat | Lam Exp | App Exp Exp
```

For any natural number $n$ the game `expGame` $n$ asks questions of expressions whose free variables are in the range 0 to $n-1$.

```
expGame :: Nat → Game Exp
expGame 0 = appLamG 0
expGame n = Split (Iso ask bld) (rangeGame 0 (n-1)) (appLamG n)
  where ask (Var i)   = Left i
        ask e         = Right e
        bld (Left i)  = Var i
        bld (Right e) = e
```

If $n$ is zero, then the expression cannot be a variable, so `expGame` immediately delegates to `appLamG` that deals with expressions known to be non-variables. Otherwise, the game is `Split` between variables (handled by `rangeGame` from Section 2) and non-variables (handled by `appLamG`). The auxiliary game `appLamG` $n$ works by splitting between application and lambda nodes:

```
appLamG n = Split (Iso ask bld) (prodGame (expGame n) (expGame n))
                                (expGame (n+1))
 where ask (App e1 e2)    = Left (e1,e2)
```

```
    ask (Lam e)         = Right e
    bld (Left (e1,e2))  = App e1 e2
    bld (Right e)       = Lam e
```

For application terms we play `prodGame` for the applicand and applicator. For the body of a $\lambda$-expression the game `expGame` $(n$`+1)` is played, incrementing $n$ by one to account for the bound variable.

Let us run the game on the expression $I\ K$ where $I = \lambda x.x$ and $K = \lambda x.\lambda y.x$.

```
> let tmI = Lam (Var 0)
> let tmK = Lam (Lam (Var 1))
> enc (expGame 0) (App tmI tmK)
[O,I,O,I,I,I,O,I]
> dec (expGame 0) it
Just (App (Lam (Var 0)) (Lam (Lam (Var 1))),[])
```

It's easy to validate by inspection the isomorphisms used in `expGame`. It's also straightforward to prove that the game is total and proper.

## 6.2 Simple types

We now move to the simply-typed $\lambda$-calculus, whose typing rules are shown in conventional form in Figure 8.

$$\frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textsc{Var} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\ \textsc{App} \qquad \frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.e : \tau_1 \to \tau_2}\ \textsc{Lam}$$

Fig. 8. Simply-typed $\lambda$-calculus

In Haskell, we define a data type `Ty` for types and `Exp` for expressions, differing from the untyped language only in that $\lambda$-abstractions are annotated with the type of the argument:

```
data Ty = TyNat | TyArr Ty Ty deriving (Eq, Show)
data Exp = Var Nat | Lam Ty Exp | App Exp Exp
```

Type environments are just lists of types, indexed de Bruijn-style. A complete program would typically be typed under some initial environment containing types for primitive constants and operations, such as `[TyNat, TyArr TyNat TyNat]` listing the types for zero and successor. For simplicity we have not included such constants in the term syntax but included a base type `TyNat` so that the set of types is inhabited. Note that, though no closed values can inhabit `TyNat`, we may still well have well-typed abstractions of type `TyArr TyNat TyNat`.

It's easy to write a function `typeOf` that determines the type of an open expression under some type environment – assuming that it is well-typed to start with.

```
type Env = [Ty]
typeOf :: Env → Exp → Ty
typeOf env (Var i) = env !! i
typeOf env (App e _) = let TyArr _ t = typeOf env e in t
typeOf env (Lam t e) = TyArr t (typeOf (t:env) e)
```

We'd like to construct a game for expressions that have type `t` under some environment `env`. If possible, we'd like the game to be *proper*. But wait: there are combinations of `env` and `t` for which no expression even exists, such as the empty environment and the type `TyNat`. We could perhaps impose an 'inhabitation' precondition on the parameters of the game. But this only pushes the problem into the game itself, with sub-games solving inhabitation problems lest they ask superfluous questions and so be non-proper. As it happens, type inhabitation for the simply-typed $\lambda$-calculus is decidable but PSPACE-complete (Sørensen & Urzyczyn, 2006), which serves to scare us off!

We can make things easier for ourselves by solving a different problem: fix the type environment `env` (as before), but instead of fixing the type as previously, we will instead fix a *pattern* of the form $\tau_1 \to \cdots \to \tau_n \to ?$ where '?' is a wildcard standing for any type. It's easy to show that for any environment `env` and pattern there exists an expression typeable under `env` whose type *matches* the pattern.

We can define such patterns using a data type `Pat`, and write a function that determines whether or not a type matches a pattern.

```
data Pat = Any | PArr Ty Pat
matches :: Pat → Ty → Bool
matches Any _ = True
matches (PArr t p) (TyArr t1 t2) = t1==t && matches p t2
matches _ _ = False
```

Now let's play some games. Types are easy:

```
tyG :: Game Ty
tyG = Split (Iso ask bld) unitGame (prodGame tyG tyG)
 where ask TyNat = Left ()
       ask (TyArr t1 t2) = Right (t1,t2)
       bld (Left ()) = TyNat
       bld (Right (t1,t2)) = TyArr t1 t2
```

To define a game for typed expressions we start with a game for variables. The function `varGame` below accepts a predicate `Ty →Bool` and an environment, and returns a game for all those indices (of type `Nat`) whose type in the environment matches the predicate.

```
varGame :: (Ty → Bool) → Env → Maybe (Game Nat)
varGame f [] = Nothing
varGame f (t:env) = case varGame f env of
 Nothing → if f t then Just (constGame 0) else Nothing
 Just g  → if f t then Just (Split succIso unitGame g)
```

```
            else Just (g +> Iso pred succ)
```

Notice that `varGame` returns `Nothing` when no variable in the environment satisfies the predicate. In all other cases it traverses the input environment. If the first type in the input environment matches the predicate *and* there is a possibility for a match in the rest of the input environment `varGame` returns a `Split` that witnesses this possible choice. It is easy to see that when `varGame` returns some game, that game will be proper.

The function `expGame` accepts an environment and a pattern and returns a game for all expressions that are well-typed under the environment and whose type matches the pattern.

```
expGame :: Env → Pat → Game Exp
-- ∀ (env:Env) (p:Pat),
--   Game { e | ∃ t, env ⊢ e : t && matches p t = true }
expGame env p
  = case varGame (matches p) env of
      Nothing → appLamG
      Just varG → Split varI varG appLamG
  where appLamG = Split appLamI appG (lamG p)
        appG = depGame (expGame env Any) $ λe →
               expGame env (PArr (typeOf env e) p)
        lamG (PArr t p) = prodGame (constGame t) $
                          expGame (t:env) p
        lamG Any = depGame tyG $ λt →
                   expGame (t:env) Any

varI = Iso ask bld where ask (Var x)   = Left x
                         ask e         = Right e
                         bld (Left x)  = Var x
                         bld (Right e) = e
appLamI = Iso ask bld
  where ask (App e1 e2)     = Left (e2,e1)
        ask (Lam t e)       = Right (t,e)
        bld (Left (e2,e1)) = App e1 e2
        bld (Right (t,e))  = Lam t e
```

The `expGame` function first determines whether the expression can possibly be a variable, by calling `varGame`. If this is not possible (case `Nothing`) the game proceeds with `appLamG` that will determine whether the non-variable expression is an application or a λ-abstraction. If the expression can be a variable (case `Just varG`) then we may immediately `Split` with `varI` by asking if the expression is a variable or not – if not we may play `appLamG` as in the first case. The `appLamG` game uses `appLamI` to ask whether the expression is an application, and then plays game `appG`; or a λ-abstraction, and then plays game `lamG`. The `appG` performs a *dependent composition*: After playing a game for the argument of the application, it binds the argument value to `e` and plays `expGame` for the function value, using the type of `e` to create a pattern for the function value. Correspondingly, the `ask` function of `appLamI` returns `Left (e2,e1)` for applications `App e1 e2` (the converse holds for `bld`) precisely because the code for the argument of the application `e2` precedes the code for the function `e1`. The `lamG` game analyses the pattern argument. If it is an arrow pattern we play a composition of the constant game for the type given by the pattern with the game for the body of the $\lambda$-abstraction in the extended environment. On the other hand, if the pattern is `Any` we first play game `tyG` for the *argument type*, bind the type to `t` and play `expGame` for the body of the abstraction using `t` to extend the environment.

That was it! Let's test `expGame` on the example expression from Section 1: $\lambda x$:`Nat`.$\lambda y$:`Nat`.$x$.

```
> let ex = Lam TyNat (Lam TyNat (Var 1))
> enc (expGame [] Any) ex
[O,I,O,O,I,I,O]
> dec (expgame [] Any) it
Just (Lam TyNat (Lam TyNat (Var 1)),[])
```

Compare the code with that obtained in the introduction. A perfect match – we have been using the same question scheme!

Finally we can show properness and totality.[4]

*Proposition 1*
For all patterns $p$ and environments $\Gamma$, the game `expGame` $\Gamma$ $p$ is proper and total for the set of expressions $e$ such that $\Gamma \vdash e : \tau$ and $\tau$ matches the pattern $p$. The codec induced by this game is therefore complete.

*Non-proper games for programs.* Given the effort we went to in order to obtain a *complete* codec, it's worth considering whether we can avoid the bother of 'patterns' at the expense of losing completeness. Given any environment and type we will construct a game for expressions typeable in that environment with that type. The function `expGameCheck` below does that.

```
-- ∀ (env:Env) (t:Ty), Game { e | env ⊢ e : t }
expGameCheck :: Env → Ty → Game Exp
expGameCheck env t
  = case varGame (== t) env of
      Nothing → appLamG t
      Just varG → Split varI varG (appLamG t)
  where appLamG TyNat
          = appG +> Iso (λ(App e1 e2)→(e2,e1))
                        (λ(e2,e1)→App e1 e2)
        appLamG (TyArr t1 t2)
          = let ask (App e1 e2)     = Left (e2,e1)
```

[4] Since we do not have `expGame` in Coq, we've only shown this on paper, hence it's a Proposition and not a Theorem.

```
                ask (Lam t e)       = Right e
                bld (Left (e2,e1)) = App e1 e2
                bld (Right e)       = Lam t1 e
            in Split (Iso ask bld) appG (lamG t1 t2)
        appG = depGame (expGame env Any) $ λe →
               expGameCheck env (TyArr (typeOf env e) t)
        lamG t1 t2 = expGameCheck (t1:env) t2
```

Similarly to `expGame`, `expGameCheck` first determines whether the expression can be a variable or not and uses the variable game or the `appLamG` next. The `appLamG` game in turn pattern matches on the input type. If the input type is `TyNat` the we know that the expression can't possibly be a λ-abstraction and hence play the `appG` game. On the other hand, if the input type is an arrow type `TyArr t1 t2` then the expression may be either application or abstraction. The application game `appG` as before plays a game for the argument of an application, binds it to `e` and recursively calls `expGameCheck` using the type of `e`. Interestingly we use `expGame env Any` to determine the type of the argument – alternatively we could perform a dependent composition where the first thing would be to play a game for the argument type, and subsequently use that type to play a game for the argument and the function. The `lamG` game is straightforward.

There are no *obvious* empty types in this game – why is it non proper? Consider the case when the environment is empty and the expected type is `TyNat`. According to `expGameCheck` the game to be played will be the `appG` game for applications. But there can't be *any* closed expressions of type `TyNat` to start with, and the game can't possibly have any leaves – something that we failed to check. We've asked a silly question (by playing `appG`) on an uninhabited type!

In other words the `expGameCheck` game is non-proper and hence its codec is not complete – not every bit 'counts'! On the other hand it's definitely a useful game and enjoys all other properties we've been discussing in this paper. Interestingly, there is a way to convert non-proper games to proper games in many cases and we return to this problem in Section 8.

### *6.3 Beyond simple types*

So far we have made complete codecs for the untyped and simply-typed lambda calculus. What about other language features and richer types, such as recursion, algebraic datatypes, polymorphism, even dependent types?

In this section we consider a modest extension: the provision of constants, at top-level, with closed polymorphic types. Given appropriate additions to the syntax of types, it's thereby possible to support language constructs such as tuples and algebraic datatypes, as their introduction and elimination forms can be supported via such constants *e.g.* `pair`: $\forall\alpha\beta.\alpha \to \beta \to \alpha \times \beta$ for constructing pairs. It's then a small step to supporting open polymorphic types and ML-style *let* polymorphism. Going beyond this – to System F, for example – whilst maintaining completeness of the encoding is an open problem.

Suppose we have a type syntax that includes type variables, say $\tau ::= \alpha \mid \texttt{int} \mid \tau \times \tau \mid \tau \to \tau$, represented by the datatype shown below:

```
data Ty = TyVar Nat | TyInt | TyArr Ty Ty | TyProd Ty Ty
```

Type schemes written $\forall\vec{\alpha}.\tau$ are assumed to be closed, and with quantified variables $\vec{\alpha}$ all occurring, in that order, in $\tau$. This lets us omit the quantifier prefix from the encoding. Environments $\Gamma$ now map variables to closed type schemes. The type system itself is identical to that of Figure 8 except that the rule for variables includes explicit instantiation of bound type variables:

$$\frac{x{:}\forall\vec{\alpha}.\tau \in \Gamma}{\Gamma \vdash x_{\vec{\tau}} : \tau[\vec{\tau}/\vec{\alpha}]}\ \textsc{Inst}$$

The corresponding game for variables must encode the instantiation $\vec{\tau}$ in addition to the index of $x$ in the environment. But to be a complete codec, it must *not* redundantly encode any type in $\vec{\tau}$ that is already determined by the pattern that is matched by the instantiated type. For example, suppose we wish to encode $\texttt{pair}_{\texttt{int},\texttt{int}\to\texttt{int}}$ and we have in our hand the pattern $\texttt{int} \to ?$. Given the type scheme $\forall\alpha\beta.\alpha \to \beta \to \alpha \times \beta$ for `pair`, we know from the pattern that $\alpha$ must be instantiated to `int` and so we should omit this from the encoding, only recording the instantiation of $\beta$, namely $\texttt{int} \to \texttt{int}$.

The match operation used in `varGame` has the following signature:

```
matches :: Pat → Ty → Maybe [Game Ty]
```

Then `matches p t` returns `Nothing` if type scheme `t` does not match `p`, and returns `Just gs` if it does match, with `gs` providing a game for each of the types in the instantiation.

Construction of this game is delicate. Suppose again that `p` is $\texttt{int} \to ?$ and we wish to find a type *scheme* that matches this pattern. The type scheme $\forall\alpha\beta.\alpha \to (\beta \to \texttt{int})$ clearly does match, but it *requires* the variable $\alpha$ to be instantiated to `int` and leaves $\beta$ undetermined. In this case `matches p t` would return `Just [constGame IntTy, tyGame]`.

## 7 Compression

All the coding schemes described thus far have ignored the expected *distribution* of values in the domain. Even a codec that is complete does not necessarily lead to *compact* encodings in practice. For example, the unary encoding of natural numbers does not waste bits, yet is optimal only for natural numbers distributed according to the probabilities $p(0) = \frac{1}{2}, p(1) = \frac{1}{4}, p(2) = \frac{1}{8}$ and so on.

In this section we consider two well-known *compression* schemes, Huffman coding and arithmetic coding. For Huffman, we present a function that constructs a game from a predetermined frequency distribution, and also an *adaptive* scheme in which the codec is updated according to symbols already seen. Arithmetic coding is more challenging, as it is not a prefix code in the sense that we use here. Here we sketch an approach based on building frequency information into the structure of the games themselves.

### *7.1 Huffman codes*

To implement Huffman coding we will need a type for frequencies and for *priority queues*. We use the following definitions:

```
type Frequency = Int
type PQ a = [(Frequency,a)]
```

A priority queue is a list of pairs, assigning integer frequencies to elements, maintained in increasing order of frequency. We provide an easy to implement interface:

```
newItem :: Frequency → a → PQ a → PQ a
incItem :: Eq a ⇒ a → PQ a → PQ a
```

Function `newItem` adds a new item to a priority queue, and `incItem` increments the frequency of an already present element.

*Static Huffman.* Our algorithm is going to work as follows. We maintain a priority queue of type `PQ (Set a, Game a)` whose elements consist of a set of values of type `a` paired with a game for the type corresponding to exactly that set. Our idea now is to pick the two elements with the lowest frequencies, combining them by taking the union of the (disjoint) sets and combining their games to produce a game for the union. The priority queue is then updated, and the process is repeated until the priority queue contains only a single set and game for elements of that set. The code is as follows:

```
bldHuff :: Ord a ⇒ PQ (Set a, Game a) → Game a
bldHuff [(_,(_,g))] = g
bldHuff ((w1,(s1,g1)):(w2,(s2,g2)):q)
  = bldHuff $ newItem w (s, Split iso g1 g2) q
    where iso = splitIso (λx → member x s1)
          w = w1 + w2
          s = s1 `union` s2
```

We assume that the priority queue is not empty to start with. In the second line of the definition, we pick the sets `s1` and `s2` with the lowest frequencies, `w1` and `w2` respectively. We create the set `s` as the union of two disjoint sets and a new frequency `w`, which is the sum of their frequencies. Now, from game `g1` for set `s1` and game `g2` for set `s2` we need to create a game for the union `s`. But that's easy! We simply have to introduce a `Split` node, where the question to be asked is whether an element `x`, belonging to the union `s`, belongs in `s1` or not.

We are almost there: given a priority queue of type `PQ a` that assigns frequencies to distinct values, we creating a priority queue of type `PQ (Set a, Game a)` for disjoint singleton sets and constant games, and then call our recursive `bldHuff` function to build the game for all values:

```
huffman :: Ord a ⇒ PQ a → Game a
huffman q = bldHuff [ (w, (singleton x, constGame x)) | (w,x) ← q ]
```

It's then easy to build a game for sequences of values that assumes a fixed distribution for each value in the sequence:

```
staticHuff :: Ord a ⇒ PQ a → Game [a]
staticHuff dist = listGame (huffman dist)
```

Let's test this out using two distributions for letters: a uniform one, and the standard Scrabble$^{TM}$ distribution:

```
uniform :: PQ Char
uniform = zip (repeat 1)
          " ABCDEFGHIJKLMNOPQRSTUVWXYZ"

scrabble :: PQ Char
scrabble = zip
           [1,1,1,1,1,2,2,2,2,2,2,2,2,2,3,4,4,4,4,5,6,6,6,8,9,9,12]
           "ZXQKJYWVPMHFCBGUSLD TRNOIAE"
```

As might be expected, the Scrabble distribution beats the uniform one, even on a short but well-known quotation:

```
> let tobe = "TO BE OR NOT TO BE THAT IS THE QUESTION"
> length (enc (staticHuff uniform) tobe)
235
> length (enc (staticHuff scrabble) tobe)
204
```

*Dynamic Huffman.* We now turn to *adaptive* Huffman codes (also known as *dynamic*), where the frequency table (and hence the coding scheme) is updated each time we encounter a new character in our input string. That's remarkably easy as well, thanks to our dependent composition combinator:

```
dynamicHuff :: Ord a ⇒ PQ a → Game [a]
dynamicHuff q = Split listIso unitGame $
                depGame (huffman q) (λx → dynamicHuff (incItem x q))
```

In `dynamicHuff`, either the list is empty, or there is a head and tail, in which case we play `depGame`, using `huffGame q` to encode the head element, and recurse on the tail. However, notice that instead of `q`, we use `incItem q`. This ensures that we dynamically update the frequency table as we read an element – and as a consequence the coding scheme itself.

As expected, the following test reveals the differences:

```
> length (enc (dynamicHuff uniform) tobe)
214
> length (enc (dynamicHuff scrabble) tobe)
199
```

Even when the initial distribution is uniform, the adaptive algorithm does nearly as well as the static algorithm on the Scrabble distribution (and on a much longer piece of text it would be much closer); but when supplied with the Scrabble distribution to start with, it emerges as overall winner.

## *7.2 Arithmetic coding*

Given a particular distribution of values, Huffman coding is the best we can do, but only if the probabilities are negative powers of two, such as $\frac{1}{2}$ or $\frac{1}{8}$. Consider a two-element type such as $\mathbb{B}$: Huffman coding must assign one-bit codes to F and to T, even if F occurs 90% of the time and T occurs 10% of the time. If we are encoding a sequence of values of the same type then we can improve compression by clumping values together, in effect doing Huffman on a product, such as $\mathbb{B} \times \mathbb{B} \times \mathbb{B}$. Then we might represent FFF by a one-bit code 0, and TTT by a much longer code 11101; we can in fact generate just these codes using `huffman`. If we take this to 'the limit', then we will achieve an optimal code for any probability distribution.

There is a better way of achieving the same compression ratio, called *arithmetic coding*. The idea is very elegant: take a representation of the real interval $[0, 1)$, and divide it up amongst the values according to their probabilities. In our example, F would occupy the interval $[0, 0.90)$ and T would occupy $[0.90, 1.00)$. A sequence of values is encoded by successively dividing subintervals in the same way. So in our example, FFF would represented by $[0, 0.729)$ and TTT by $[0.999, 1.0)$. Once the entire input has been processed, the output is any number that uniquely identifies the current interval, *i.e.*, any number inside the current interval. The number is encoded as a binary expansion.

It's not possible to use our games for the machinery of arithmetic coding. In any case, there are many efficient implementations already, including a slick one in Haskell (Bird & Gibbons, 2003). But the tree-like dividing of intervals does suggest a generalization of our games: attach probabilities, or frequencies, to the branches of `Split` nodes, and then use arithmetic coding as a 'back-end' in new versions of the `enc` and `dec` functions. Instead of emitting 0 or 1, we divide the reals according to the frequencies.

Games are easily modified:

```
data Game t where
  Single :: ISO t () → Game t
  Split :: ISO t (Either t1 t2) → Int → Game t1 →
                                  Int → Game t2 → Game t
```

We can then define a *biased* game for booleans, as above:

```
biasedBool = Split boolIso 1 unitGame 9 unitGame
```

We may test it on vectors of length 6:

```
> enc (vecGame biasedBool 6) [False,False,False,False,False,False]
[]
> enc (vecGame biasedBool 6) [False,False,False,False,False,True]
[O,I,I]
> enc (vecGame biasedBool 6) [False,False,True,False,False,True]
```

```
[O,O,I,I,O,I]
> enc (vecGame biasedBool 6) [True,False,True,False,False,True]
[O,O,O,O,O,O,I]
> enc (vecGame biasedBool 6) [True,True,True,True,True,True]
[O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O]
```

As we increase the number of `True` values in the list, the code gets longer and longer, reflecting the relative rarity of `True` in our distribution. Also observe that arithmetic coding is *not* a prefix code; indeed, the representation for six successive `False` values is the empty list, reflecting the fact that the number 0 is sufficient to uniquely identify the interval $[0, 0.9^6)$.

It's fairly straightforward to generalize our binary trees to $n$-ary ones, with isomorphisms on nodes of the form $T \cong T_1 + \cdots + T_n$. It would of course be valuable to model the distributions *adaptively*, somehow updating the frequencies in the game as it is played. Also, most work on arithmetic coding is confined to streams of symbols, whereas in our representation we get arithmetic coding 'for free' for whatever types are modelled using games. Of particular interest is the case of typed programs (Cheney, 2000). All of these features are the subject of future research.

## 8 Filtering games

Much of the ingenuity required to construct *proper* games and hence *complete* codecs comes from the necessity to work in a subset of a domain, such as lists without duplicates (for sets), or well-typed terms. In this section we consider the possibility of taking a game for an easily-described domain and *filtering* it by some predicate to obtain a game for the subset.

*Non-proper filtering.* We first study a filtering function that results in a non-proper game. Using `voidGame` from Section 4.4 we can write `filterGame`, which accepts a game and a predicate on `t` and returns a game for those elements of `t` that satisfy the predicate.

```
filterGame :: (t → Bool) → Game t → Game t
-- ∀ (p : t → Bool), Game t → Game { x | p x }
filterGame p g@(Single (Iso _ bld)) =
 if p (bld ()) then g else voidGame
filterGame p (Split (Iso ask bld) g1 g2)
 = Split (Iso ask bld) (filterGame (p ∘ bld ∘ Left)  g1)
                       (filterGame (p ∘ bld ∘ Right) g2)
```

It works by inserting `voidGame` in place of all singleton nodes that do not satisfy the filter predicate. We may, for instance, filter a game for natural numbers to obtain a game for the even natural numbers.

```
> enc (filterGame even binNatGame) 2
[I,I,O]
> dec (filterGame even binNatGame) [I,I,O]
```

```
(2,[])
```

Naturally, since the game is no longer proper, decoding can fail:

```
> dec (filterGame even binNatGame) [I,O,I,O,O,I,I,I,I]
(*** Exception: Input too short
```

Moreover, for the above bitstring, no suffix is sufficient to convert it to a valid code – we have entered the `voidGame` non-proper world.

What is so convenient about the non-proper `filterGame` implementation? First, the structure of the original encoding is intact with only some codes being removed. Second, it avoids hard inhabitation questions that may involve theorem proving or search.

*Proper finite filtering.* Now let's recover properness, with the following variant on filtering:

```
filterFinGame :: (t → Bool) → Game t → Maybe (Game t)
-- ∀ (p : t → Bool), Game t → option (Game { x | p x })
filterFinGame p g@(Single (Iso _ bld)) =
  if p (bld ()) then Just g else Nothing
filterFinGame p (Split iso@(Iso ask bld) g1 g2)
  = case (filterFinGame (p ∘ bld ∘ Left) g1,
            filterFinGame (p ∘ bld ∘ Right) g2) of
      (Nothing, Nothing)    → Nothing
      (Just g1', Nothing)   → Just $ g1' +> iso1
      (Nothing, Just g2')   → Just $ g2' +> iso2
      (Just g1', Just g2') → Just $ Split iso g1' g2'
  where fromLeft  (Left x)  = x
        fromRight (Right x) = x
        iso1 = Iso (fromLeft  ∘ ask) (bld ∘ Left )
        iso2 = Iso (fromRight ∘ ask) (bld ∘ Right)
```

The result of applying `filterFinGame` is of type `Maybe (Game t)`. If *no* elements in the original game satisfy the predicate, then `filterFinGame` returns `Nothing`, otherwise it returns `Just` a game for those elements of `t` satisfying the predicate. In contrast to `filterGame`, though, `filterFinGame` preserves proper-ness: if the input game is proper, then the result game is too. It does this by eliminating `Split` nodes whose subgames would be empty.

There is a limitation, though, as its name suggests: `filterFinGame` works only on *finite* games. This can be inferred from the observation that `filterFinGame` explores the game tree in a depth-first manner. Nevertheless, for such finite games we can use it profitably to obtain efficient encodings:

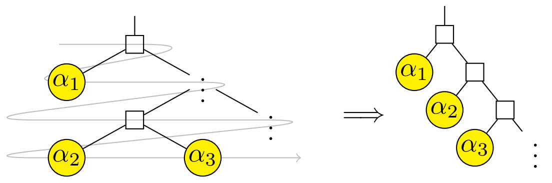
```
>  enc (fromJust (filterFinGame even (rangeGame 0 7))) 4
[I,O]
```

Compare this to the original encoding before filtering:

```
> enc (rangeGame 0 7) 4
[I,O,O]
```

*Proper infinite filtering.* What about infinite domains, as is typically the case for recursive types? Can we implement a filter on games that produces proper games for such types?

The answer is yes, if we are willing to drastically change the original encoding that the game expressed, and *if* that original game has infinitely many leaves that satisfy the filter predicate. Here is the idea, not given here in detail for reasons of space, but implemented in the accompanying code as function `filterInfGame`: perform a breadth-first traversal of the original game, and each time you encounter a new singleton node (that satisfies the predicate) insert it into a right-spined tree:



The ability to become proper in this way can help us recover proper games for simply-typed expressions of a given type in a given environment, from the weaker games that `expGameCheck` of Section 6.2 produces, *if* we have a precondition that there exists one expression of the given type in the given environment. If there exists one expression of the given type in the given environment, there exist infinitely many, and hence the `expGameCheck` game has infinitely many inhabitants. Consequently it is possible to rebalance it in the described way to obtain a proper game for simply-typed expressions!

```
--expGameCheckProper env t
--  = filterInfGame (const True) (expGameCheck env t)
```

## 9 Discussion

### *9.1 Practicality*

There is no reason to believe that the game-based approach is suitable only for theoretical investigations but not for 'real' implementations. To test this hypothesis we intend to apply the technique to a reasonably-sized compiler intermediate language such as Haskell Core (Sulzmann *et al.*, 2007) or .NET CIL (ECMA, 2006). (We've already created an every-bit-counts codec for ML-style let polymorphism.)

Determining the space complexity of games is somewhat tricky: as we navigate down the tree, pointers to thunks representing *both* the left and the right subtrees are kept around, although only one of two pointers is relevant. An optimization would involve embedding the next game to be played on *inside* the isomorphism, by making the `ask` functions return not only a split but also, for each alternative (left or right), a next game to play on. Hence only the absolutely relevant parts of the game would be kept around during encoding and decoding. This representation could then be subject to the optimizations described in stream fusion work (Coutts *et al.*, 2007). For this paper though our goal has been to explain the semantics of games and not their optimization and hence we used the easier-to-grasp definition of a game as just a familiar tree datatype.

It's also worth noting that the encoding and decoding functions can be specialized by hand for particular games, eliminating the game construction completely. For a trivial example, consider inlining `unaryNatGame` into `enc`, performing a few simplifications, to obtain the following code:

```
encUnaryNat x = case x of 0   → I : []
                          n   → O : encUnaryNat (n-1)
```

### *9.2 Test generation.*

Test generation tools such as Quickcheck (Claessen & Hughes, 2000) are a potential application of game-based decoding, since generating random bitstrings amounts to generating programs. As a further direction for research, we would like to examine how the programmer could affect the distribution of the generated programs, by tweaking the questions asked during a game.

### *9.3 Program development and verification in Coq.*

Our attempts to encode everything in this paper in Coq tripped over Coq's limited support for co-recursion, namely the requirement that recursive calls be *guarded* by constructors of coinductive data types (Bertot & Casteran, 2004). In many games for recursive types, the recursive call was under a use of a combinator such as `prodGame`, which was itself guarded. Whereas it is easy to show on paper that the resulting co-fixpoint is well-defined (because it is productive), Coq does not admit such definitions. On the positive side, using the proof obligation generation facilities of `Program` (Sozeau, 2006) was a very pleasant experience. Our Coq code in many cases has been a slightly more verbose version of the Haskell code (due to the more limited type inference), but the isomorphism obligations could be proven on the side. Our overall conclusion from the experience is that Coq itself *can become* a very effective development platform but it would benefit from better support for more general patterns of recursion, co-recursion, and type inference.

## 10 Related work

Our work has strong connections to Kennedy's pickler combinators (Kennedy, 2004). There, a codec was represented by a pair of encoder and decoder functions, with codecs for complex types built from simple ones using combinators. The basic round-trip property (Enc/Dec) was considered informally, but stronger properties were not studied. Before developing the game-based codecs, we implemented by hand encoding and decoding functions for the simply-typed $\lambda$-calculus. Compared to the game presented in Section 6, the code was more verbose – partly because out of necessity both encoder and decoder used the same 'logic'. In our opinion, games are more succint representations of codecs, and their correctness is easier to verify than the correctness of codecs written with pickler combinators, as games require only *local* reasoning about isomorphisms. Note that other related work (Duan *et al.*, 2005) identifies and formally proves similar round-trip properties for encoders and decoders in several encryption schemes.

One can think of games as yet another technique for datatype-generic programming (Gibbons, 2007), where one of the most prominent applications is generic marshalling and unmarshalling. Many of the approaches to datatype-generic programming (Hinze *et al.*, 2006) are based on the structural representations of datatypes, typically as fixpoints of functors consisting of sums and products. It is straightforward to derive automatically a default 'structural' game for recursive and polymorphic types. On the other hand, games are convenient for expressing *semantic* aspects of the values to be encoded and decoded, such as naturals in a given range. Moreover, the state of a game and therefore the codes themselves can be modified as the game progresses, which is harder (but not impossible, perhaps through generic views (Holdermans *et al.*, 2006)) in datatype-generic programming techniques. Finally, our definition of advanced games is somewhat low-level because we have to explictly map from the constructors of a datatype to left and right injections – it would be interesting to determine whether some generic structural representation could help program our games at an even higher-level of abstraction.

Another related area of work is data description languages, which associate the semantics of types to their low-level representations (Fisher *et al.*, 2006). The interpetation of a datatype *is* a coding scheme for values of that datatype. There, the emphasis is on *avoiding* manually having to write encode and decode functions. Our goal is slightly different; more related to the properties of the resulting coding schemes and their verification rather than the ability to automatically derive encoders and decoders from data descriptions.

Though we have not seen games used for writing and verifying encoders and decoders, tree-like structures have been proposed as representations of mathematical functions. For instance, some related work (Ghani *et al.*, 2009) represents continuous functions on streams as binary trees. In our case, thanks to the embedded isomorphisms, the tree structures represent at the same time both the encode and the decode functions.

The idea of compact codes for (syntactically) well-formed programs is itself old, dating at least back to the work of Contla (1985) and Cameron (1988). More recently, researchers have investigated codes for typed program compression, some claiming high compression ratios for every-bit-counts (and hence tamper-proof) codes for low-level bytecode (Haldar *et al.*, 2002; Franz *et al.*, 2002). Although that work is not formalized, it is governed by the design principle of only asking questions that 'make sense'. That is precisely what our properness property expresses, which provably leads to every bit counts codes. Some of these ideas have also been recently applied for compression of Javascript code as abstract syntax trees (Burtscher *et al.*, 2010).

Finally, closely related is the idea behind oracle-based checking (Necula & Rahul, 2001) in proof carrying code (Necula & Lee, 1998). The motivation there is to eliminate proof search for untrusted software and reduce the size of proof encodings. In oracle-based checking, the bitstring oracle guides the proof checker in order to eliminate search and unambiguously determine a proof witness. Results report an improvement of a *factor* of 30 in the size of proof witnesses compared to their naïve syntactic representations. Although not explicitly stated in this way, oracle-based checking really amounts to a game for well-typed terms in a variant of LF. Oracle-based coding appeared again in recent work (Nielsen & Henglein, 2011), where it is used for the efficient representation of regular expression parse trees. The codes obtained there for parse trees represent the "choices" that a parsing algorithm would perform to associate a regular expression with a certain parse tree. It is an interesting direction for future work to express the set of bit-coded parse trees as a game for parse trees.

## Acknowledgments

The authors appreciated the lively discussions on this topic at the 'Type Systems Wrestling' event held weekly at MSR Cambridge. Special thanks to Johannes Borgström and the anonymous ICFP 2010 reviewers for their helpful feedback. Thanks to Chung-Kil Hur for helping solve some problems in our Coq formalization. Finally we would like to thank the reviewers of the journal submission for their many suggestions; they encouraged us to include Coq code and theorems, which we believe strengthens the work considerably.

## References

Bertot, Y., & Casteran, P. (2004). *Interactive Theorem Proving and Program Development.* Springer-Verlag.

Bird, R., & Gibbons, J. (2003). Arithmetic coding with folds and unfolds. *Pages 1–26 of:* Jeuring, J., & Peyton Jones, S. (eds), *Advanced Functional Programming 4*. Lecture Notes in Computer Science, vol. 2638. Springer-Verlag.

Burtscher, M., Livshits, B., Sinha, G., & Zorn, B. 2010 (June). JSZap: Compressing JavaScript code. *Proceedings of the USENIX Conference on Web Application Development.*

Cameron, Robert D. (1988). Source encoding using syntactic information source models. *IEEE Transactions on Information Theory*, **34**(4), 843–850.

Cheney, J. (2000). Statistical models for term compression. *Page 550 of: DCC '00: Proceedings of the Conference on Data Compression.* Washington, DC, USA: IEEE Computer Society.

Claessen, K., & Hughes, J. (2000). Quickcheck: a lightweight tool for random testing of Haskell programs. *Pages 268–279 of: ICFP '00: Proceedings of the fifth ACM SIGPLAN International Conference on Functional Programming.* New York, NY, USA: ACM.

Contla, Jose F. (1985). Compact coding of syntactically correct source programs. *Softw. Pract. Exper.*, **15**(July), 625–636.

Coutts, D., Leshchinskiy, R., & Stewart, D. (2007). Stream fusion: from lists to streams to nothing at all. *Pages 315–326 of: ICFP '07: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming.* New York, NY, USA: ACM.

Duan, J., Hurd, J., Li, G., Owens, S., Slind, K., & Zhang, J. (2005). Functional correctness proofs of encryption algorithms. *Pages 519–533 of: Logic for Programming, Artificial Intelligence and Reasoning (LPAR).* LNCS, vol. 3835. Springer.

ECMA. (2006). *Standard ECMA-335: Common Language Infrastructure (CLI).*

Elias, P. (1975). Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, **21**(2), 197–203.

Fisher, K., Mandelbaum, Y., & Walker, D. (2006). The next 700 data description languages. *SIGPLAN not.*, **41**(1), 2–15.

Franz, M., Haldar, V., Krintz, C., & Stork, C. H. 2002 (March). *Tamper-proof annotations by construction.* Tech. rept. 02-10. Dept of Information and Computer Science, University of California, Irvine.

Ghani, N., Hancock, P., & Pattinson, D. (2009). Representations of stream processors using nested fixed points. *Logical Methods in Computer Science*, **5**(3).

Gibbons, J. (2007). Datatype-generic programming. *Chap. 1, pages 1–71 of:* Backhouse, R., Gibbons, J., Hinze, R., & Jeuring, J. (eds), *Datatype-generic programming.* LNCS, vol. 4719. Berlin, Heidelberg: Springer.

Haldar, V., Stork, C. H., & Franz, M. (2002). The source is the proof. *Pages 69–73 of: NSPW '02: Proceedings of the 2002 workshop on new security paradigms.* New York, NY, USA: ACM.

Hinze, R., Jeuring, J., & Löh, A. (2006). Comparing approaches to generic programming in Haskell. *Spring School on Datatype-Generic Programming.*

Holdermans, S., Jeuring, J., Löh, A., & Rodriguez, A. (2006). Generic views on data types. *Pages 209–234 of: In proceedings of the 8th International Conference on Mathematics of Program Construction, MPC06, volume 4014 of LNCS.* Springer.

Kennedy, A. J. (2004). Functional Pearl: Pickler Combinators. *Journal of Functional Programming*, **14**(6), 727–739.

MacKay, D. J. C. (2003). *Information Theory, Inference and Learning Algorithms.* Cambridge University Press.

Necula, G. C., & Lee, P. (1998). The design and implementation of a certifying compiler. *Pages 333–344 of: PLDI '98: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation.* New York, NY, USA: ACM.

Necula, G. C., & Rahul, S. P. (2001). Oracle-based checking of untrusted software. *Pages 142–154 of: POPL'01: Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* New York, NY, USA: ACM.

Nielsen, Lasse, & Henglein, Fritz. (2011). Bit-coded regular expression parsing. *Proc. 5th Int'l Conf. on Language and Automata Theory and Applications (LATA).* Lecture Notes in Computer Science (LNCS). Springer.

Salomon, D. (2008). *A Concise Introduction to Data Compression.* Undergraduate Topics in Computer Science. Springer.

Sørensen, M. H., & Urzyczyn, P. (2006). *Lectures on the Curry-Howard Isomorphism, Volume 149 (Studies in Logic and the Foundations of Mathematics).* New York, NY, USA: Elsevier Science Inc.

Sozeau, M. (2006). Subset coercions in Coq. *Pages 237–252 of: Selected papers from the International Workshop on Types for Proofs and Programs (TYPES '06).* Springer.

Sulzmann, M., Chakravarty, M., & Peyton Jones, S. (2007). System F with type equality coercions. *ACM Workshop on Types in Language Design and Implementation (TLDI).* ACM.