Functional Pearl: Every Bit Counts

Dimitrios Vytiniotis
Microsoft Research, Cambridge, U.K.
dimitris@microsoft.com

Andrew J. Kennedy
Microsoft Research, Cambridge, U.K.
akenn@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 λ-calculus. 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.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features; E.4 [*CODING AND INFORMATION THEORY*]: [Data compaction and compression]

General Terms Design, Languages, Theory

1. Introduction

Let's 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 λ? No.
> You *must be* $\lambda x{:}\texttt{Nat}.\lambda y{:}\texttt{Nat}.x$. *You're right!*

[1] A closed program in the simply-typed λ-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 α-equivalence. We have deliberately impoverished the language for simplicity of presentation; in practice there would also be constants, primitive operations, and perhaps other constructs.

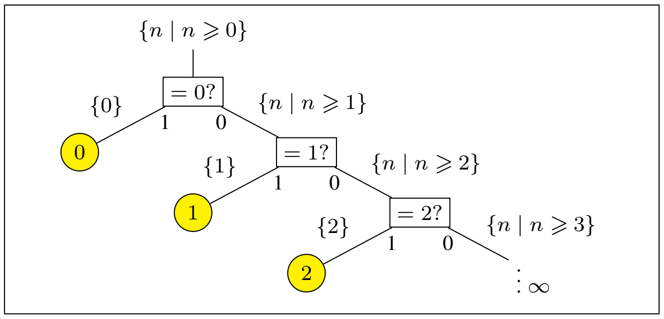
From the answer to the first question, we know that the program is not a function application. Moreover, the program is closed, and so it *must* be a λ-abstraction; hence we proceed to ask new questions about the argument type and body. We continue asking questions until we have identified the program. 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 [10, 13] and compact proof witnesses in proof carrying code [19]. 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 endodings.

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-answer games for encoding and decoding: a novel way to think about and program codecs (Section 2). We build simple 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.
- 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 3).
- We develop more sophisticated codecs for abstract types such as sets and multisets, making crucial use of the invariants associated with such types (Section 4).
- We build games for untyped and simply-typed terms that yield every-bit-counts coding schemes (Section 5). 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 a typed language that has been proven correct.
- We discuss filters on games (Section 6). Finally, we discuss future developments and present connections to related work (Sections 7 and 8).



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

Figure 1: Unary game for naturals

We will be using Haskell (for readability, familiarity, and executability) but the paper is accompanied by a partial Coq formalization (for correctness) downloadable from:

`http://research.microsoft.com/people/dimitris/`

The correctness and compactness properties of our coding schemes follow *by construction* in our Coq development, and by very localized reasoning in our Haskell code. We make use of infinite structures, utilizing laziness in Haskell (and co-induction in Coq), but the code should adapt to call-by-value languages through the use of thunks.

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* 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*: a bitstring uniquely determines a value. Moreover,

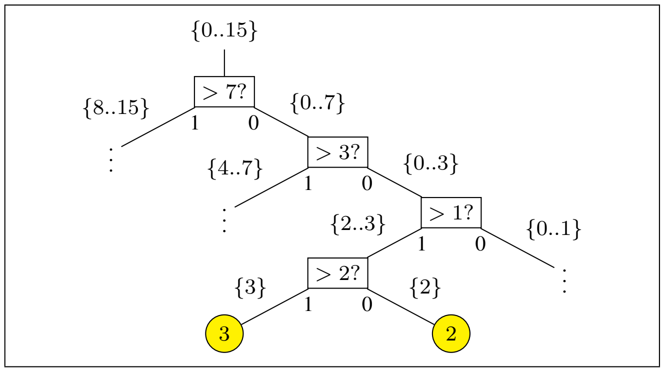


Figure 2: Binary game for $\{0..15\}$

the one-question-at-a-time nature of games ensures that codes are *prefix-free*: no code is the prefix of any other valid code [20].

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 of 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. In Section 3 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 τ_1 representing the domain of the left subtree (corresponding to answering *yes* to the question) and τ_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, in Haskell!

2.1 Games in Haskell

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

```
data Game :: * → * 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 `()`, Haskell's unit type. The nodes carry a representation of an isomorphism between `t` and `Either t1 t2` (Haskell's sum type), and two subtrees of type `Game t1` and `Game t2`.[2]

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

```
-- (Iso to from) must satisfy
--   left inverse:  from ∘ to = id
--   right inverse: to ∘ from = id
data ISO t s = Iso { to :: t → s, from :: s → t }
```

In our Coq formalization, the `ISO` type also records *proofs* of the left inverse and right inverse properties.

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:

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

Given a `Game t`, here is an encoder for `t`:

```
enc :: Game t → t → [Bit]
enc (Single _) x = []
enc (Split (Iso ask _) g1 g2) x
  = case ask x of Left x1  → I : enc g1 x1
                  Right x2 → O : enc g2 x2
```

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] → (t, [Bit])
dec (Single (Iso _ bld)) str = (bld (), str)
dec (Split _ _ _) [] = error "Input too short"
dec (Split (Iso _ bld) g1 g2) (I : xs)
  = let (x1, rest) = dec g1 xs
    in (bld (Left x1), rest)
dec (Split (Iso _ bld) g1 g2) (O : xs)
  = let (x2, rest) = dec g2 xs
    in (bld (Right x2), rest)
```

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` raises an exception.[3] 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 return the unique value in `t` by applying the inverse map of the isomorphism on `()`. 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 `O` or `I`, building a value of `t` using the `bld` function of the isomorphism gadget.

2.2 Number games

These simple definitions already suffice for a range of numeric encodings. We make the Haskell definition

```
type Nat = Int
```

to document that our integers are non-negative. Where the Haskell type system isn't rich enough to express precise invariants, we will put Coq types in comments, lifted directly from our Coq development.

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

[3] We could alternatively have `dec` return `Maybe (t,[Bit])`; this is indeed what our Coq formalization does.

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

```
singleIso :: a → ISO a ()
-- ∀ x:a, ISO {z | z = x} unit
singleIso x = Iso (const ()) (const x)
```

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

```
splitIso :: (a → Bool) → ISO a (Either a a)
-- ∀ p:a→bool, ISO a ({x|p x = true}+{x|p x = false})
splitIso p = Iso ask bld
  where ask x = if p x then Left x else Right x
        bld x = case x of Left y → y; Right y → y
```

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

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

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

```
parityIso :: ISO Nat (Either Nat Nat)
parityIso = Iso
  (λn → if even n then Left(n `div` 2)
                  else Right(n `div` 2))
  (λx → case x of Left m → m*2; Right m → m*2+1)
```

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

```
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
```

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

```
depListIso :: ISO [t] (Nat,[t])
-- ISO (list t) { n:nat & t^n }
depListIso = Iso ask bld where ask xs = (length xs, xs)
                               bld (n,xs) = xs
```

Figure 3: Some useful isomorphisms

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

```
geNatGame :: Nat → Game Nat
-- ∀ k:nat, Game { x | x ⩾ k }
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 `Singleton` 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, 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` $(n-1)$, that capture both the *yes*/*no* 'answer' to the question 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]
> dec unaryNatGame [O,O,I]
Just (2,[])
```

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

```
rangeGame :: Nat → Nat → Game Nat
-- ∀ m n : nat, Game { x | m ⩽ x && x ⩽ n }
rangeGame m n | m == n = Single (singleIso m)
rangeGame m n = Split (splitIso (λx → x > 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]
(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 0 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 3 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)
```

We can test this game; for example:

```
> enc binNatGame 8
[O,O,O,I,O,I,I]
> dec binNatGame [O,O,O,I,O,I,I]
Just (8,[])
> enc binNatGame 16
[O,O,O,I,O,I,O,I,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 the next section we explain how the extra bits result in *prefix codes*, a property that our methodology is designed to validate by construction.

The accompanying Haskell code gives additional examples of games for natural numbers, including Elias codes [8], as well as codes based on prime factorization.

2.3 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.

```
constGame :: t → Game t
-- ∀ (k:t), Game { x | x=k }
constGame k = Single (singleIso k)
```

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

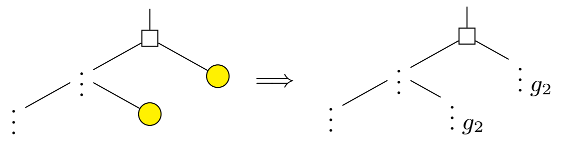
```
(+>) :: Game t → ISO s t → Game s
(Single j) +> i      = Single (i `seqI` j)
(Split j g1 g2) +> i = Split  (i `seqI` j) g1 g2
```

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 4 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 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`. Graphically, if g_1 looks like the tree on the left, below, composing it with g_2 produces the tree on the right.



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)
```

$A \cong A$	`idI :: ISO a a`
$A \cong B \Rightarrow B \cong A$	`invI :: ISO a b →ISO b a`
$A \cong B \wedge B \cong C \Rightarrow A \cong C$	`seqI :: ISO a b →ISO b c →ISO a c`
$A \cong B \wedge C \cong D \Rightarrow A \times C \cong B \times D$	`prodI :: ISO a b →ISO c d →ISO (a,c) (b,d)`
$A \cong B \wedge C \cong D \Rightarrow A + C \cong B + D$	`sumI :: ISO a b →ISO c d →ISO (Either a c) (Either b d)`
$A \times B \cong B \times A$	`swapProdI :: ISO (a,b) (b,a)`
$A + B \cong B + A$	`swapSumI :: ISO (Either a b) (Either b a)`
$A \times (B \times C) \cong (A \times B) \times C$	`assocProdI :: ISO (a,(b,c)) ((a,b),c)`
$A + (B + C) \cong (A + B) + C$	`assocSumI :: ISO (Either a (Either b c)) (Either (Either a b) c)`
$\mathbf{1} \times A \cong A$	`prodLUnitI :: ISO ((),a) a`
$A \times \mathbf{1} \cong A$	`prodRUnitI :: ISO (a,()) a`
$A \times (B + C) \cong (A \times B) + (A \times C)$	`prodRSumI :: ISO (a,Either b c) (Either (a,b) (a,c))`
$(B + C) \times A \cong (B \times A) + (C \times A)$	`prodLSumI :: ISO (Either b c,a) (Either (b,a) (c,a))`

Figure 4: Isomorphism combinator signatures

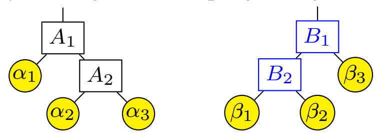
If the game for `t` is a singleton node, then we play `g2`, which is the game for `s`. However, that will return a `Game s`, whereas we'd like a `Game (t,s)`. But from the type of the `Single` constructor we know that `t` is the unit type `()`, and so we *coerce* `g2` to the appropriate type using combinators from Figure 4 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 4.

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

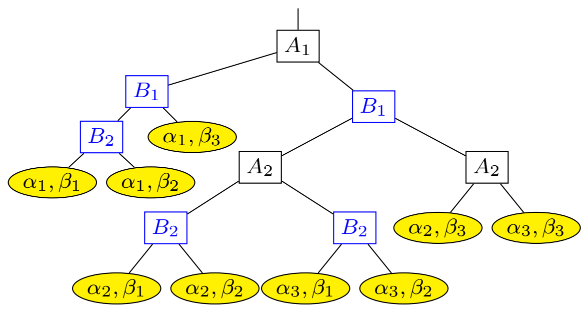
```
listGame :: Game t → Game [t]
listGame g =
  Split listIso unitGame (prodGame g (listGame g))
```

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. 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)
-- Game t → (∀ x:t, Game(s x)) → Game {x:t & s x}
depGame (Single iso) f =
  f (from iso ()) +> prodI iso idI ‘seqI‘ prodLUnitI
depGame (Split iso g1a g1b) f
  = Split (prodI iso idI ‘seqI‘ prodLSumI)
          (depGame g1a (f ∘ from iso ∘ Left))
          (depGame g1b (f ∘ from iso ∘ 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.

Lists, revisited. We can use `depGame` to create an alternative encoding for lists. Suppose we are given a function

```
vecGame :: Game t → Nat → Game [t]
-- Game t → ∀ n:nat, Game t^n
```

that builds a game for lists of the given length. Its definition should be straightforward and we leave it as an exercise for the reader. We can then define a game for lists paired with their length, and use the isomorphism `depListIso` from Figure 3 to derive a new game for lists, as follows:

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

3. Properties of games

Pearly code is all very well, but is it correct? In this section we study the formal properties of game-derived codecs, proving basic correctness and termination results, and also the *every bit counts* property of the title. All theorems have been proved formally using the Coq proof assistant.

3.1 Correctness

The following round-trip property follows directly from the 'left inverse' property of isomorphisms embedded inside the games.

LEMMA 1 (Enc/Dec). *Suppose* g : `Game t` *and* x : `t`. *If* `enc` $g\ x = \ell$ *then* `dec` $g\ (\ell +\!\!+ \ell_s) = (x, \ell_s)$.

The lemma asserts that if x encodes to a bitstring ℓ, then the decoding of any extension of ℓ returns x together with the extension.

The literature on coding theory [20] emphasizes the essential property of codes being *unambiguous*: no two values are assigned the same code. This follows directly from Lemma 1.

COROLLARY 1 (Unambiguous codes). *Suppose* g : `Game t` *and* v, w : `t`. *If* `enc` $g\ v = \ell$ *and* `enc` $g\ w = \ell$ *then* $v = w$.

A stronger property that implies unambiguity is *prefix-freedom*: no prefix of a valid code can itself be a valid code. For prefix codes, we can stop decoding at the first successfully decoded value: no 'look-ahead' is required. This property also follows from Lemma 1, or can be proved directly from the definition of `enc`.

COROLLARY 2 (Prefix encoding). *Suppose* g : `Game t` *and* v, w : `t`. *If* `enc` $g\ v = \ell$ *and* `enc` $g\ w = \ell +\!\!+ \ell_s$ *then* $v = w$.

It is worth pausing for a moment to return briefly to the game `binNatGame` from Section 2.1. Observe that the 'standard' binary encoding for natural numbers *is not* a prefix code. 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 downside are the inserted 'terminator' bits that double the size of the encoding (but keeping it $\Theta(\log n)$).

3.2 Termination

A close inspection of Lemma 1 reveals that the property is conditional on the *termination* of the encoder. 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.

Here is a problematic example of a somewhat funny game for the type `Maybe Nat`, appearing in Figure 5. At step i, the game asks whether the value in hand is `Some` i, or any other value in the type `Maybe Nat`. Notice that when asked to encode a value `Nothing` 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.

DEFINITION 1 (Totality). *A game* g *of type* `Game t` *is* total *iff for every value* x *of type* `t`, *there exists a finite path* $g \rightsquigarrow x$, *where* $\rightsquigarrow$ *is inductively defined below:*

$$\frac{}{\texttt{Single (Iso } a\ b) \rightsquigarrow b\ ()} \qquad \frac{g_1 \rightsquigarrow x_1}{\texttt{Split (Iso } a\ b)\ g_1\ g_2 \rightsquigarrow b\ (\texttt{Left } x_1)}$$

$$\frac{g_2 \rightsquigarrow x_2}{\texttt{Split (Iso } a\ b)\ g_1\ g_2 \rightsquigarrow b\ (\texttt{Right } x_2)}$$

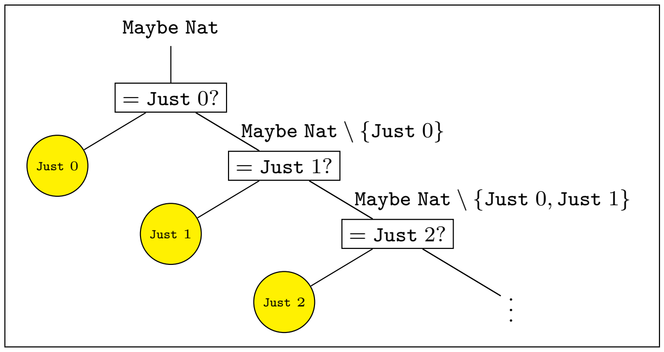


Figure 5: Game for optional naturals

The reader can check that, with the exception of the game in Figure 5, the games presented so far are total; furthermore the combinators on games preserve totality.

LEMMA 2 (Termination). *Suppose* g : `Game t`. *If* g *is total then* `enc` g *terminates on all inputs.*

3.3 Compactness

Lemma 1 guarantees basic correctness of game-based codes.[4] But we can go further, and show how to construct codecs for which *every bit counts*, i.e. there are no 'wasted' bits.

Consider the following trivial codec for booleans:

```
boolGame :: Game Bool
boolGame = Split boolIso unitGame unitGame
```

It encodes `False` as 0, as `True` as 1. You can't do better than that!

Now consider a codec in which both 00 and 01 code for `False`, and 10 and 11 code for `True`. The second bit of this code is wasted, as the first bit uniquely determines the value. Fortunately, correct construction of game 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 another round-trip property that follows directly from the 'right inverse' property of isomorphisms.

LEMMA 3 (Dec/Enc). *Suppose* g : `Game t`. *If* `dec` $g\ \ell = (x, \ell_s)$ *then there exists* ℓ_p *such that* `enc` $g\ x = \ell_p$ *and* $\ell_p +\!\!+ \ell_s = \ell$.

Injectivity of decoding is a simple corollary.

COROLLARY 3 (Non-redundancy). *Suppose* `dec` $g\ \ell_1 = (x, [])$ *and* `dec` $g\ \ell_2 = (x, [])$. *Then* $\ell_1 = \ell_2$.

Unfortunately non-redundancy doesn't tell us that *every bit counts*. Consider a slight variation on the wasteful encoding of booleans above in which `True` is encoded as 11 and `False` as 00, and 01 and 10 are simply *invalid*. This corresponds to a question-answer game in which the question *Are you* `True`*?* is asked twice. We can write such a game, as follows:

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

badBoolGame :: Game Bool
badBoolGame = Split (splitIso id)
  (Split (splitIso id) (constGame True) voidGame)
  (Split (splitIso id) voidGame (constGame False))
```

[4] But, to be fair, sometimes lossy coding may be acceptable; for instance in video codecs.

It may take a little head-scratching to work out what's going on: the question expressed with `splitIso id` asks whether a boolean value is `True` or `False` and goes `Left` or `Right` respectively. But in both branches we ask the same question again, though we're now in a singleton set. Here's a session that illustrates the `badBoolGame` behaviour:

```
> enc badBoolGame False
[O,O]
> enc badBoolGame True
[I,I]
> dec badBoolGame [O,I]
(False,*** Exception: Input too short
> dec badBoolGame [I,O]
(True,*** Exception: Input too short
```

The first question asked by the game effectively partitions the booleans into {`False`} and {`True`}. But these are singletons, so any further questions would not reveal further information. If we do ask a question, using `Split`, then one branch must be dead, *i.e.* have a domain that is not inhabited – hence the use of `voidGame` in the code.

For domains more complex than `Bool`, such non-revealing 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. Equivalently, we make the following definition.

DEFINITION 2 (Proper games). *A game g of type `Game t` is* proper *iff for every subgame g' of type `Game s`, type `s` is inhabited.*

It is immediate that `voidGame` is not a proper game and consequently `badBoolGame` is not proper either.

Codecs associated with proper games have a very nice property that justifies the slogan *every bit counts*: every possible bitstring either decodes to a unique value, or is the prefix of such a bitstring.

LEMMA 4 (Every bit counts). *Let g be a proper and total `Game t`. Then, if* dec g ℓ *fails then there exists ℓ_s and a value x of type `t` such that* enc g $x = \ell +\!\!+\, \ell_s$.

The careful reader will have observed that this lemma requires that the game be not only proper, but also *total*. Consider the following variation of `binNatGame` from Section 2.2.

```
badNatGame :: Game Nat
badNatGame = 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 Lemma 4 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.

3.4 Summary

Here is what we have learned in this section.

- Games constructed from valid isomorphisms give rise to codes that are unambiguous, prefix-free, non-redundant, and which satisfy a basic round-trip correctness property.
- The encoder terminates if and only if the game is total.
- 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 and λ-terms.

4. Sets and multisets

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 [12, 14] to generate games (and thereby codecs) automatically for such types.

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 representing sets of natural numbers using lists. We know (a) that duplicate elements do not occur, and (b) that the order doesn't matter when considering a list-as-a-set. We could use `listGame binNatGame` for this type. It would satisfy the basic round-tripping property (Enc/Dec); 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 represent sets and multisets efficiently. First, we consider the specific case of sets and multisets of natural numbers, for which we can hand-craft a 'delta' encoding in which every bit counts. Next, we show how for arbitrary types we can use an ordering on values induced by the game for the type to construct a game for sets of elements of that type.

4.1 Hand-crafted games

How can we encode the multiset $\{3, 6, 5, 6\}$? We might start by ordering the values to obtain the *canonical* representation $[3, 5, 6, 6]$. But now imagine encoding this using a vanilla list of natural numbers game `listGame binNatGame`: when encoding the second element, we would be wasting the codes for values 0, 1, and 2, as none of these values 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 and add the difference.

We can apply the same 'delta' idea for sets, except that the delta is smaller by one, taking account of the fact that the difference between successive elements must be non-zero.

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

```
diff minus [] = []
diff minus (x:xs) = x : diff' x xs
  where diff' base [] = []
        diff' base (x:xs) = minus x base : diff' x xs

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 [Nat]
natMultisetGame g =
  listGame g +> Iso (diff (-) ∘ sort) (undiff (+))

natSetGame :: Game Nat → Game [Nat]
natSetGame g =
  listGame g +> Iso (diff (λ x y → x-y-1) ∘ sort)
                    (undiff (λ x y → x+y+1))
```

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

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

As expected, the encoding is more compact than a vanilla list representation. Observe that here the round-trip property holds *up to equivalence* of lists when interpreted as multisets: encoding `[3,6,5,6]` and then decoding it results in an equivalent but not identical value `[3,5,6,6]`.

4.2 Generic games

That's all very well, but 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 do sets:

```
setGame :: Game a → Game [a]
setGame g = setGame' g +> Iso (sortByGame g) id
  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, once a value is determined plays the game having removed all smaller elements from it.[5]

5. 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 games for sets 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.

5.1 No types

First let's play a game for the untyped λ-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 λ-expression the game `expGame` $(n+1)$ is played, incrementing n by one to account for the bound variable.

Let's 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
(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.

[5] The `$` notation is just Haskell syntactic sugar that allows applications to be written with fewer parentheses: `f (h g)` can be written as `f $ h g`.

$$\frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textsc{Var} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\ \textsc{App}$$

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.e : \tau_1 \rightarrow \tau_2}\ \textsc{Lam}$$

Figure 6: Simply-typed λ-calculus

5.2 Simple types

We now move to the simply-typed λ-calculus, whose typing rules are shown in conventional form in Figure 6.

In Haskell, we define a data type `Ty` for types and `Exp` for expressions, differing from the untyped language only in that λ-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. 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 λ-calculus is decidable but PSPACE-complete [21], 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 $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow ?$ 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 – it 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. 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 expression for the body of the λ-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
(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.[6]

PROPOSITION 1. *For all patterns p and environments Γ, the game* `expGame` Γ p *is proper and total for the set of expressions e such that $\Gamma \vdash e : \tau$ and τ matches the pattern p.*

5.3 Stronger non-proper games for typed expressions

Let us be brave now and return to the original problem. Given any environment and type we will construct a game for expressions typeable in that environment with that type. As we have noted above, obtaining a proper game (and hence an every bit counts encoding) is difficult, but we can certainly obtain a game easily without having to implement a type inhabitation solver if we give up properness. 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)
                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!

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

In other words the `expGameCheck` game is non-proper and hence violates the every bit counts property. On the other hand it's definitely a useful game and enjoys all other properties we've been discussing in this paper. Happily, there is a way to convert non-proper games to proper games in many cases and we return to this problem in the next section.

6. Filtering games

Non-proper filtering. Sometimes it's convenient *not* to be proper. Using `voidGame` from Section 3.3 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 with 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:

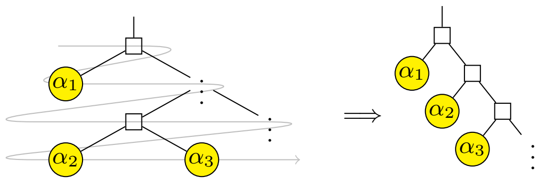
```
> enc (fromJust (filterFinGame even (rangeGame 0 7))) 4
[I,O]
```

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 5.3 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 (λ_ ⟶ True) (expGameCheck env t)
```

7. Discussion

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 [23] or .NET CIL [7]. (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 [5]. 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+1 ⟶ O : encUnaryNat n
```

Compression. For reasons of space, we have compressed away any discussion of classic techniques such as Huffman coding. In the accompanying code, however, the reader can find a function `huffGame` that accepts a list of frequencies associated with elements of type `t` and returns a `Game t` constructed using the Huffman technique. Adaptive (or dynamic) Huffman encoding is achieved using just two more lines of Haskell!

Investigation of other compression techniques using games remains future work. In particular, we would like to integrate arithmetic coding, for which slick Haskell code already exists [2].

It would also be interesting to make use of statistical models in our games for typed programs [3], producing codes that are even more compact than is attained purely through the use of type information.

Test generation. Test generation tools such as Quickcheck [4] are a potential application of game-based decoding, since generating random bitstrings amounts to generating programs. As a further direction for research, we wold like to examine how the programmer could affect the distribution of the generated programs, by tweaking the questions asked during a game.

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 [1]. 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` [22] 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.

8. Related work

Our work has strong connections to Kennedy's pickler combinators [16]. 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 λ-calculus. Compared to the game presented in Section 5, 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 are easier to verify, requiring only *local* reasoning about isomorphisms. Note that other related work [6] 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 [12], where one of the most prominent applications is generic marshalling and unmarshalling. Many of the approaches to datatype-generic programming [14] 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 [15]) in datatype-generic programming techniques.

Another related area of work is data description languages, which associate the semantics of types to their low-level representations [9]. 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. Ghani *et al.* [11] represent 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.

Other researchers have investigated typed program compression, claiming high compression ratios for every-bit-counts (and hence tamper-proof) codes for low-level bytecode [13, 10]. 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. Also closely related is the idea behind oracle-based checking [19] in proof carrying code [18]. 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 some game for well-typed terms in a variant of LF.

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 for his helpful feedback, and to the anonymous reviewers for many helpful suggestions.

References

[1] Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development*. Springer-Verlag, 2004.

[2] R. Bird and J. Gibbons. Arithmetic coding with folds and unfolds. In J. Jeuring and S. Peyton Jones, editors, *Advanced Functional Programming 4*, volume 2638 of *Lecture Notes in Computer Science*, pages 1–26. Springer-Verlag, 2003. Code available at `http://www.comlab.ox.ac.uk/oucl/work/jeremy.gibbons/publications/arith.zip`.

[3] J. Cheney. Statistical models for term compression. In *DCC '00: Proceedings of the Conference on Data Compression*, page 550, Washington, DC, USA, 2000. IEEE Computer Society.

[4] K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of Haskell programs. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN International Conference on Functional Programming*, pages 268–279, New York, NY, USA, 2000. ACM.

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

[6] J. Duan, J. Hurd, G. Li, S. Owens, K. Slind, and J. Zhang. Functional correctness proofs of encryption algorithms. In *Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 3835 of *LNCS*, pages 519–533. Springer, 2005.

[7] ECMA. Standard ECMA-335: Common language infrastructure (CLI), 2006.

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

[9] K. Fisher, Y. Mandelbaum, and D. Walker. The next 700 data description languages. *SIGPLAN Not.*, 41(1):2–15, 2006.

[10] M. Franz, V. Haldar, C. Krintz, and C. H. Stork. Tamper-proof annotations by construction. Technical Report 02-10, Dept of Information and Computer Science, University of California, Irvine, March 2002.

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

[12] J. Gibbons. Datatype-generic programming. In R. Backhouse, J. Gibbons, R. Hinze, and J. euring, editors, *Datatype-Generic Programming*, volume 4719 of *LNCS*, chapter 1, pages 1–71. Springer, Berlin, Heidelberg, 2007.

[13] V. Haldar, C. H. Stork, and M. Franz. The source is the proof. In *NSPW '02: Proceedings of the 2002 workshop on New security paradigms*, pages 69–73, New York, NY, USA, 2002. ACM.

[14] R. Hinze, J. Jeuring, and A. Löh. Comparing approaches to generic programming in Haskell. In *Spring School on Datatype-Generic Programming*, 2006.

[15] S. Holdermans, J. Jeuring, A. Löh, and A. Rodriguez. Generic views on data types. In *In T. Uustalu, editor, Proceedings of the 8th International Conference on Mathematics of Program Construction, MPC06, volume 4014 of LNCS*, pages 209–234. Springer, 2006.

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

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

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

[19] D. Salomon. *A Concise Introduction to Data Compression*. Undergraduate Topics in Computer Science. Springer, 2008.

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

[21] M. Sozeau. Subset coercions in Coq. In *Selected papers from the International Workshop on Types for Proofs and Programs (TYPES '06)*, pages 237–252. Springer, 2006.

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