Functional Pearl: Every Bit Counts

Dimitrios Vytiniotis
Microsoft Research
dimitris@microsoft.com

Andrew Kennedy
Microsoft Research
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 \( \lambda \)-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.

1. Introduction

Let’s play a guessing game:

\[
\begin{align*}
\text{I am a simply-typed program.} & \quad \text{Can you guess which one?} \\
\text{Are you a function application?} & \quad \text{No.} \\
\text{You must be a function.} & \quad \text{Is your argument a \texttt{Nat}? Yes.} \\
\text{Is your body a variable?} & \quad \text{No.} \\
\text{Is your body a function application?} & \quad \text{No.} \\
\text{It must be a function.} & \quad \text{Is its argument a \texttt{Nat}? Yes.} \\
\text{Is its body a variable?} & \quad \text{Yes.} \\
\text{Is it bound by the nearest \texttt{\lambda}?} & \quad \text{No.} \\
\text{You must be } \lambda x: \texttt{Nat}. \lambda y: \texttt{Nat}. x & \quad \text{You’re right!}
\end{align*}
\]

From the answer to the first question, we know that the program is not a function application. Moreover, the program is closed, and therefore it can only 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 which questions constitute our game we’ve thereby built an encoder for programs. By interpreting a bit sequence as answers to that same game, we have a decoder. If we choose our questions carefully, we can make sure that no two programs are assigned the same code (no ambiguity), no two codes identify the same program (no redundancy), and moreover, any bit sequence either has a prefix that is a valid code, or is a prefix of a valid code (no junk). No junk justifies the phrase in the title ‘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]. This paper crystallizes and formalizes the key intuition behind both those works: question-and-answer games.

More specifically, in this paper we show how to program games, from which we create codecs for numbers, lists, and sets, building up to every-bit-counts codes for terms of the simply-typed \( \lambda \)-calculus. Our contributions are as follows:

- We introduce games for encoding and decoding: a novel way to think about and program codecs (Section 2). We build simple games for numeric types, and provide combinators that construct complex games from simpler ones, producing correct-by-construction coding schemes for structured, recursive, polymorphic, and indexed types.
- 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 games for abstract types such as sets and multisets, making crucial use of the invariants associated with such types. (Section 4)
- We build question-answer games for simply-typed terms that yield unambiguous and non-redundant codes. In addition, we give an every-bit-counts coding scheme (Section 5) for simply-type terms. To our knowledge, this is the first provably such coding scheme for a typed language. Finally, we give discussion and connections to related work. (Sections 6 and 7)

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 non-ambiguity and non-redundancy 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright © ACM [to be supplied] ... $10.00
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, 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-free encoding of \( n \in \mathbb{N} \), 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-free encoding of \( n \in \mathbb{N} \).

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 \( \{ n \mid n \geq 0 \} \). 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 \( \{ n \mid n \geq 1 \} \). 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 the range \([0, 15]\). 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 \( n \in [0, 15] \) whether \( n \in [8, 15] \) or \( n \in [0, 7] \), splitting the range into two disjoint parts. 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 yes and all right edges with no, 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 \( n \) zeros 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.

Let’s dive straight in, with a data type for games:

```haskell
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 with domain `t`, whose leaves built with `Single` represent singletons and whose nodes built with `Split` 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 sub-games of type `Game t1` and `Game t2`.

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

```haskell
-- (Iso to from) must satisfy invariants
-- to o from = id and from o to = id
data ISO t s = Iso { to :: t -> s, from :: s -> t }
```

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:

```haskell
type Bit = Int -- 0 or 1
```

2 The type variables `t1` and `t2` are *existential variables*, not part of vanilla Haskell 98, but supported by all modern Haskell compilers.
Given a Game t, here is an encoder for t:

\[
\text{enc} :: \text{Game } t \rightarrow t \rightarrow \text{[Bit]}
\]
\[
\text{enc} (\text{Single } _\_ ) x = []
\]
\[
\text{enc} (\text{Split } (\text{Iso } \_ \_ ) g1 g2) x =
\]
\[
\begin{cases}
\text{let } x1, x2 = \text{dec } g2 x2 \\
\text{in } \text{dec } g1 x1 \\
\text{if } x \text{ then } 0 : \text{enc } g2 x2
\end{cases}
\]

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 be split to a value of either t1 or t2 for some t1 and t2. Depending on the answer we output 1 or 0 and continue playing either the sub-game g1 or g2.

A decoder is also simple to write:

\[
\text{dec} :: \text{Game } t \rightarrow \text{[Bit]} \rightarrow (t, \text{[Bit]})
\]
\[
\text{dec} (\text{Single } _\_ ) \text{str } = (\text{str}, \text{str})
\]
\[
\text{dec} (\text{Split } _\_ ) [] = \text{error } \text{"Input too short"}
\]
\[
\text{dec} (\text{Split } (\text{iso } \_ \_ ) g1 g2) (1:xs) =
\]
\[
\begin{cases}
\text{let } x1, x2 = \text{dec } g2 xs \\
\text{in } (\text{bld } (\text{Left } x1), \text{rest})
\end{cases}
\]
\[
\text{dec} (\text{Split } (\text{iso } \_ \_ ) g1 g2) (0:xs) =
\]
\[
\begin{cases}
\text{let } x2, rest = \text{dec } g2 xs \\
\text{in } (\text{bld } (\text{Right } x2), \text{rest})
\end{cases}
\]

The decoder accepts a game Game t and a bitstring of type [Bit]. If the input bitstring is too short to decode a value then dec raises an error indicating this\(^3\). Otherwise it returns a decoded value of type t and the suffix of the input list that was not consumed. If the game is Single, then dec can 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 but the input list is empty then dec raises an error. Otherwise, depending on the first bit, dec decodes the rest of the bitstring using either sub-game g1 or g2, building a value of t using the bld function of the isomorphism gadget.

Hopefully the way that type isomorphisms are used in Split is now clear. When encoding a value, we ask questions of the data using the forward map of the isomorphism to get answers of the form Left x or Right y, that capture both the yes/no 'answer' to the question and data with which to continue playing the game. When decoding, we apply the inverse map of the isomorphism to build data with Left x or Right x as determined by the next bit in the input stream.

A trivial game for booleans expresses this most directly, utilizing the isomorphism between Bool and Either () () (or in mathematical notation, \(\equiv 1 + 1\)).

\[
\text{unitGame} :: \text{Game } ()
\]
\[
\text{unitGame} = \text{Single } (\text{Iso } () () (\lambda () \rightarrow ()))
\]
\[
\text{boolIso} :: \text{ISO } \text{Bool } (\text{Either } () () )
\]
\[
\text{boolIso } =
\]
\[
\text{Iso } (\lambda b \rightarrow \text{if } b \text{ then Left } x \text{ else Right } x)
\]
\[
(\rightarrow \text{case } x \text{ of Left } x \rightarrow \text{True } ; \text{Right } x \rightarrow \text{False})
\]
\[
\text{boolGame} :: \text{Game } \text{Bool}
\]
\[
\text{boolGame} = \text{Split } \text{boolIso } \text{unitGame } \text{unitGame}
\]

### 2.2 Warm-up: number games

These simple definitions are already enough to write games for a range of numeric encodings.

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

**Unary games for naturals** The following function accepts an integer k and returns a game for natural numbers greater than or equal to k. In Haskell it is difficult to express such invariants using types so we will be commenting our functions with their 'real' more expressive types in the rest of this paper.

\[
\text{-- geNatGame } k \text{ returns a game for } \{ n: \text{Nat } | n \geq k \}
\]
\[
\text{geNatGame } k = \text{Split } \text{iso } (\text{constGame } k) (\text{geNatGame } (k+1))
\]
\[
\text{where iso : ISO } \text{Nat } (\text{Either } \text{Nat } \text{Nat})
\]
\[
\text{iso } = \text{Iso } \text{ask } \text{bld}
\]
\[
\text{-- Precondition of ask } x: x \geq k
\]
\[
\text{ask } x = \text{if } x = k \text{ then } \text{Left } x \text{ else } \text{Right } x
\]
\[
\text{bld } (\text{Left } x) = x
\]
\[
\text{bld } (\text{Right } x) = x
\]

What is constGame k? It's simply a singleton game for some data which is exactly equal to k:

\[
\text{constGame } k = \text{Single } (\text{Iso } (\text{const }()) (\text{const } k))
\]

Notice that our definition of geNatGame exactly matches the tree of Figure 1. Each recursive call to geNatGame corresponds to a rectangular node in the figure. Each call to constGame corresponds to a circular node in the figure.

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

\[
> \text{enc } (\text{geNatGame } 0) 3
\]
\[
[0,0,0,1]
\]
\[
> \text{enc } (\text{geNatGame } 0) 2
\]
\[
[0,0,1]
\]
\[
> \text{dec } (\text{geNatGame } 0) [0,0,1]
\]
\[
\text{Just } (2,[1])
\]

There is yet another, simpler, game for naturals based on their unary encoding. This time the game just asks if a number n is zero or not: if the answer is yes then we are done, otherwise we play the very same game for the predecessor of n.

\[
\text{succIso } : \text{ISO } \text{Nat } (\text{Either } () \text{Nat})
\]
\[
\text{succIso } = \text{Iso } \text{ask } \text{bld}
\]
\[
\text{where ask } 0 = \text{Left } ()
\]
\[
\text{ask } (n+1) = \text{Right } n
\]
\[
\text{bld } (\text{Left } ()) = 0
\]
\[
\text{bld } (\text{Right } n) = n+1
\]
\[
\text{unaryNatGame } :: \text{Game } \text{Nat}
\]
\[
\text{unaryNatGame } = \text{Split } \text{succIso } \text{unitGame } \text{unaryNatGame}
\]

All the magic lies in succIso. The ask function asks whether the number is 0. If yes, it returns () on the left. If not it returns the predecessor of a on the right. It is easy to see by inspection that ask and bld form an isomorphism between Nat and Either () Nat, or \(\equiv 1 + N\).

**The range game for naturals** How about the range encoding for natural numbers, sketched in Figure 2? Easy:

\[
\text{-- Precondition for rangeGame } k1 \text{ k2: } k1 \leq k2
\]
\[
\text{rangeGame } k1 k2 = \text{Split } \text{iso } \text{bld} \text{ g1 g2}
\]
\[
\text{where g1 } = \text{rangeGame } (n+1) k2
\]
\[
g2 = \text{rangeGame } k1 m
\]
\[
\text{ask } x = \text{if } x > m \text{ then } \text{Left } x \text{ else } \text{Right } x
\]
\[
\text{bld } (\text{Left } x) = x
\]
\[
\text{bld } (\text{Right } x) = x
\]
\[
m = (k1 + k2) \text{ 'div' } 2
\]

The game proceeds by keeping two numbers k1 and k2 as its state that corresponds to the range of numbers it is currently dealing.
with, very much like the tree in Figure 2. Again, the magic is in the isomorphism gadget. Function ask goes left or right depending on whether the current value is greater than, or less than or equal to the median value.

The binary game for naturals The range encoding results in a logarithmic coding scheme, but only works for naturals in a given range. Can we give a general logarithmic scheme for arbitrary naturals? Yes, and here is the protocol: we first ask if the number is 0 or not. If yes, we are done. If not, we ask whether it is divisible by 2 or not. After playing the game for the quotient, in the first case we multiply by 2 and in the second we multiply by 2 and add 1. In other words, there is a isomorphism \( N \cong N + N \), via division by 2. Here is the code:

```haskell
binNatGame :: Game Nat
binNatGame = Split succIso unitGame divG
  where divG = Split (Iso ask bld) binNatGame binNatGame
        ask n | even n   = Left (n `div` 2)
               | otherwise = Right (n `div` 2)
        bld (Left m) = 2*m
        bld (Right m) = 2*m+1
```

We can test this game; for example:

```haskell
> enc binNatGame 8
[0,0,0,1,0,1,1]
> dec binNatGame [0,0,0,1,0,1,1]
Just (8,[])
> enc binNatGame 16
[0,0,0,1,0,1,0,1,1]
```

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 coding scheme, but only works for naturals in a given range. Can we give a general logarithmic scheme for arbitrary naturals? Yes, and here is the protocol: we first ask if the number is 0 or not. If yes, we are done. If not, we ask whether it is divisible by 2 or not. After playing the game for the quotient, in the first case we multiply by 2 and in the second we multiply by 2 and add 1. In other words, there is a isomorphism \( N \cong N + N \), via division by 2. Here is the code:

```haskell
binNatGame :: Game Nat
binNatGame = Split succIso unitGame divG
  where divG = Split (Iso ask bld) binNatGame binNatGame
        ask n | even n   = Left (n `div` 2)
               | otherwise = Right (n `div` 2)
        bld (Left m) = 2*m
        bld (Right m) = 2*m+1
```

We can test this game; for example:

```haskell
> enc binNatGame 8
[0,0,0,1,0,1,1]
> dec binNatGame [0,0,0,1,0,1,1]
Just (8,[])
> enc binNatGame 16
[0,0,0,1,0,1,0,1,1]
```

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 1 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. The simplest combinator, (+ >), transforms a game for \( t \) into a game for \( a \), given that \( a \) is isomorphic to \( t \).

```haskell
(+ >) :: Game t -> Game a -> Game (Either t a)
(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 3 and their implementation (and proof) is entirely straightforward.

Choice It’s dead easy to construct games for sums of two types, if we are given games for each. The \( \text{sumGame} \) combinator is so simple that it hardly has a reason to exist as a separate definition:

```haskell
sumGame :: Game t -> Game s -> Game (Either t s)
sumGame = Split idI
```

### Figure 3: Isomorphism combinator signatures

<table>
<thead>
<tr>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>prodI</td>
<td>ISO (a,b) (Either b a)</td>
</tr>
<tr>
<td>prodLUnitI</td>
<td>ISO ([],a)</td>
</tr>
<tr>
<td>prodRUnitI</td>
<td>ISO (a,[])</td>
</tr>
<tr>
<td>prodRSumI</td>
<td>ISO (a,Either b c) (Either (a,b) (a,c))</td>
</tr>
<tr>
<td>prodLSumI</td>
<td>ISO (Either b c, a) (Either (b,a) (c,a))</td>
</tr>
<tr>
<td>assocProdI</td>
<td>ISO (a,(b,c)) (a,(b,c))</td>
</tr>
<tr>
<td>assocSumI</td>
<td>ISO (Either (a,b) c) (Either a (b,c))</td>
</tr>
<tr>
<td>assocProdSumI</td>
<td>ISO (a,(b,c)) (a,(b,c))</td>
</tr>
<tr>
<td>assocSumProdI</td>
<td>ISO (Either (a,b) c) (Either a (b,c))</td>
</tr>
<tr>
<td>listIso</td>
<td>ISO [t]</td>
</tr>
<tr>
<td>listIso'</td>
<td>ISO [t] (Either () (t,[t]))</td>
</tr>
<tr>
<td>listIso'</td>
<td>ISO [t] (Either () (t,[t]))</td>
</tr>
</tbody>
</table>

Composition Suppose we are given a game \( g_1 \) of type \( \text{Game} \ t \) and a \( g_2 \) of type \( \text{Game} \ s \). How can we build a game for products \( (t,s) \)?

A simple strategy is to play \( g_1 \), the game for \( t \), and at the leaves insert copies of \( g_2 \), the game for \( s \). Graphically, if \( g_1 \) looks like the tree on the left, below, composing it with \( g_2 \) creates the structure on the right.

```haskell
prodGame :: V t s. Game t -> Game s -> Game (t,s)
prodGame (Single iso) g2 = g2 +> iso'
  where iso' :: ISO (t,s) (Either (ta,s) (tb,s))

prodGame (Split (iso::ISO t (Either ta tb)) g1a g1b) g2
  = Split iso' (prodGame g1a g2) (prodGame g1b g2)
```

The code to achieve this is given by the \( \text{prodGame} \) combinator:

```haskell
prodGame :: V t s. Game t -> Game s -> Game (t,s)
prodGame (Single iso) g2 = g2 +> iso'
  where iso' :: ISO (t,s) (Either (ta,s) (tb,s))
```

If the game for \( t \) is a singleton node, then we play \( g_2 \), which is the game for \( s \). However, that will return a \( \text{Game} \ s \), whereas we’d like a \( \text{Game} \ (t,s) \). Fortunately, we can coerce \( g_2 \) to the appropriate type since we are able to construct an isomorphism \( iso' \) between \( (t,s) \) and \( s \). Readers should be able to convince themselves that such an isomorphism can be constructed, using the available isomorphism \( iso \) of type \( \text{Game} \ t \), without even looking at the exact combinatorial definition of \( iso' \). In the case where \( \text{Game} \ t \) is a split node, we are going to simply be asking a question to a pair \( (t,s) \) that is derived from the original question to \( t \), and can give us back \( \text{Either} \ (t_a,s) (t_b,s) \) depending on what the answer from \( t \) was. This is taken care of by the isomorphism \( iso' \) which in turn uses \( iso \) (again, readers should not bother too much about the definition of \( iso' \)). Recursively, we can create the product of \( g_2 \) with the sub-games of \( g_1 \), \( g_1a \) and \( g_1b \).

Recursion What can we do with prodGame? We can use it to build more complex combinators, such as the following that can operate on lists (or streams):

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

The listGame function accepts a game for t and creates a game for lists (or streams) of t. The question asked, defined by listIso is whether the list is empty or not. If the list is empty (left sub-game) we have a singleton node. If the list is non-empty (right sub-game) we have to play the game for the element in the head of the list followed by the very same game for lists, for the tail of the list. This is just the product prodGame g (listGame g). In math notation, listIso simply expresses the isomorphism \( t^* \cong 1 + t \times t^* \).

**Composition by interleaving** Notice that prodGame g plays always bits from the first game, and when that game finishes, it plays the bits of the second game. An alternative approach would be to *interleave* the bits of the two games. Here is a graphical illustration. Suppose that we start with the games given below:

![Interleaving two games](image)

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:

ilGame :: ∀ t s. Game t → Game s → Game (t,s)
ilGame (Single iso) g2 = g2 \(→\) iso'
  where iso' :: ISO (t,s) s -- assuming ISO t ()
  iso' = prod iso id iso' prodI iso
ilGame (Split (iso :: ISO t (Either ta tb)) ga gb) g2
  where iso' :: ISO (t,s) (Either (ta,s) (tb,s))
  iso' = prod iso id iso' prodI iso
  iso' = iso' (prod iso id iso' prodI iso)
  iso' = prodI iso' prodI iso
  iso' = prodI iso' prodI iso

The resulting encoding of product values of course differs between ilGame and prodGame. The ilGame may be more convenient for combining games for infinite data structures, such as streams.

**Dependent composition** Suppose that, after having decoded a value of datatype t, we wish to play a game which depends on the particular value that has been decoded: for instance, given a game for natural numbers, and a game for lists of a particular size, we might want to create a game for arbitrary lists paired up with their size. We may do this with the help of a *dependent product* game combinator.

Ideally (in type theory), the signature of this combinator would be:

\[
\forall t \ s. \ Game t \rightarrow (x:t) \rightarrow Game (s,x) \rightarrow Game (\Sigma x:t, s x)
\]

In Haskell, we compromise with the simpler type of depGame given below:

depGame :: ∀ t s. Game t → (t → Game s) → Game (s,t)
depGame (Split (iso :: ISO t (Either ta tb)) ga gb) f
  = Split iso' (depGame ga fa) (depGame gb fb)
  where fa :: ta → Game s
  fb :: tb → Game s
  iso' :: ISO (t,s) (Either (ta,s) (tb,s))
  iso' = prod iso id ISO 'seqI' prodI iso

The definition of depGame resembles the definition of prodGame, but notice how in the Single case we call the f function on the singleton value to determine the game we must play next. In the Split case we have to create functions fa and fb to make the types match up for the recursive calls to ga and gb. Their implementation is completely determined by the types we have to construct.

How can we use depGame? It is illustrative to use it to create yet another encoding for lists. Suppose first that we are given a function

vecGame :: Game t → Nat → Game [t]

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 (hint: recurse on the list argument, use constGame in the 0 case and prodGame in the non-zero case). We may then define a game for lists paired up with their length, and use that to derive yet another game for lists, listGame’:

lengthListGame :: Game t → Game (Nat,[t])
lengthListGame g = depGame binNatGame (vecGame g)

listGame’ :: ∀ t. Game t → Game [t]
listGame’ g = lengthListGame g \(→\) ISO h j
  where h :: [t] → (Nat,[t])
  j :: (Nat,[t]) → [t]
  -- Precondition: n = length lst
  j (n,lst) = lst

3. Properties of games

Parely 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 isomorphisms embedded inside the games.\(^4\)

**Lemma 1** (Enc/Dec). Suppose \(g : Game t \land x : t\). If \(\text{enc} \ g \ x = \ell \) then \(\text{dec} g (\ell + \ell_s) = (x, \ell_s)\).

\(^4\)Strictly speaking, it follows from ‘half’ the isomorphism, namely that in an \text{iso} to \text{from}, \text{fromto} = \text{id}.
The lemma asserts that if \( x \) encodes to a bitstring \( \ell \), then the decoding of any extension of \( \ell \) 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 \( g : \text{Game} \ t \) and \( v, w : t \). If \( \text{enc} \ g \ v = \ell \) and \( \text{enc} \ g \ w = \ell \) 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 \( \text{enc} \).

**Corollary 2 (Prefix encoding).** Suppose \( g : \text{Game} \ t \) and \( v, w : t \). If \( \text{enc} \ g \ v = \ell \) and \( \text{enc} \ g \ w = \ell' \) then \( v = w \).

It is worth pausing for a moment to return briefly to the game \( \text{binNatGame} \) 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 \( \text{binNatGame} \) 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 \( \Theta(\log n) \)).

All games presented so far give rise to unambiguous prefix codes. This follows from the correct construction of isomorphisms; and in Coq, the very type of \( \text{ISO} \) forces us to formally prove the isomorphism properties and hence we derive games that are correct by construction.

### 3.2 Termination

Having unambiguous codes is an essential correctness property of an encoding\(^5\), but is it the only essential property that we care about? A close inspection of Lemma 1 reveals that the stated non-ambiguity property is conditional on the termination of the encoder. Although in traditional coding theory termination of encoding for any value is taken for granted, that is not the case in the games setting.

Here is a problematic example of a somewhat funny game for the datatype \( \text{Maybe Nat} \), appearing in Figure 4. At step-1, the game asks whether the value in hand is \( \text{Some i} \), or any other value in the datatype \( \text{Maybe Nat} \). Notice that when asked to encode a value nothing the encoder will simply play the game for ever, diverging.

\(^5\) But, to be fair, sometimes lossy coding may be desirable as well; for instance in video codecs.

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 \( \text{Game} \ t \) is total iff for every value \( x \) of type \( t \), there exists a finite path \( g \rightsquigarrow x \), where \( \rightsquigarrow \) is inductively defined below:

\[
\begin{align*}
\text{Single} (\text{Iso} \ a \ b) \rightsquigarrow b() & \quad \text{Split (Iso} \ a \ b \text{)} \ g_1 \ g_2 \rightsquigarrow b(\text{Left} \ x_1) \\
& \quad \text{Split (Iso} \ a \ b \text{)} \ g_1 \ g_2 \rightsquigarrow b(\text{Right} \ x_2)
\end{align*}
\]

The reader can easily check that the games presented so far are total and the combinators preserve totality.

**Lemma 2 (Termination).** Suppose \( g : \text{Game} \ t \). If \( g \) is total then \( \text{enc} \ g \) terminates on all inputs.

#### 3.3 Non-redundancy

Games guarantee that two values will never be assigned the same code, but what about the converse: is it possible that two codes represent the same value? Although there is nothing wrong with allowing such codes, they are arguably less efficient. Happily, the isomorphism gadgets embedded inside the games eliminate such codes.

**Lemma 3 (Dec/Enc).** Suppose \( g : \text{Game} \ t \). If \( \text{dec} \ g \ \ell = (x, \ell_x) \) then there exists \( \ell_p \) such that \( \text{enc} \ g \ x = \ell_p \) and \( \ell_p \uplus \ell_x = \ell \).

Injectivity of decoding follows then easily.

**Corollary 3 (Non-redundancy).** Suppose \( \text{dec} \ g \ \ell_1 = (x, [\]) \) and \( \text{dec} \ g \ \ell_2 = (x, [\]) \). Then \( \ell_1 = \ell_2 \).

Hence, all valid codes ‘count’ in an information-theoretic sense: all resulting codes represent different values. But, do all bits count? Can we make all bistrings valid codes?

#### 3.4 Proper games

An example of redundancy would be a codec for booleans that associates the codes \( 00 \) and \( 01 \) with \( \text{False} \) and \( 10 \) and \( 11 \) with \( \text{True} \). Corollary 3 ensures that codes can’t be ‘wasted’ in this way. But consider a coding for booleans in which \( \text{True} \) is encoded as 11 and \( \text{False} \) as 00, and 01 and 10 are plainly invalid. This corresponds to a question-answer game in which the question \( \text{Are you True?} \) is asked twice. We can write such a game, as follows:

```
-- Assumption: t is uninhabited
voidIso :: ISO t (Either t t)
voidIso = Iso Left (\x\rightarrow case x of Left v\rightarrow v; Right v\rightarrow v)
voidGame :: Game t
voidGame = Split voidIso voidGame voidGame

badBoolGame :: Game Bool
badBoolGame = Split boolIso
  (\Split(boolIso (\_\rightarrow Left () (\_\rightarrow ()))) unitGame voidGame)
  (\Split(boolIso (\_\rightarrow Right()) (\_\rightarrow ()))) voidGame unitGame)
```

It may take a little head-scratching to work out what’s going on: the question expressed with \( \text{boolIso} \) asks whether a boolean value is \( \text{True} \) or \( \text{False} \) and goes \( \text{Left} \) or \( \text{Right} \) respectively. But the following question in both cases is silly: we ask whether a unit value is indeed a unit value or it belongs in the empty set (the latter expressed with playing \( \text{voidGame} \))! Here’s a session that illustrates the \( \text{badBoolGame} \) behaviour:
The first question asked by the game effectively partitions the boolean values into \{
\text{False}\} and \{
\text{True}\}. But these are singletons, so any further questions would not reveal further information. If we do ask a question, using \text{Split}, then one branch must be dead, i.e. have a domain that is not inhabited – hence the use of \text{voidGame} in the code.

For domains more complex than \text{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 ‘\text{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 \text{no}.

We call a game \text{proper} if every isomorphism in \text{Split} nodes is a proper splitting of the domain. Equivalently, we make the following definition.

**Definition 2** (Proper games). A game \(g\) of type \(\text{Game\ t}\) is proper iff for every subgame \(g'\) of type \(\text{Game\ s}\), \(\text{type\ s}\) is inhabited.

It is immediate that \text{voidGame} is not a proper game and consequently \text{badBoolGame} is not proper either.

Codecs associated with proper games have a very nice property that justifies the slogan \text{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 \text{Game\ t}. Then, if \(\text{dec\ g\ t}\) fails then there exists \(\ell_s\) and a value \(x\) of type \(t\) such that \(\text{enc\ g\ x} = \ell + \ell_s\).

Notice though, 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.

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 \text{binNatGame} from Section 2.2.

```haskell
binNatGame' :: Game Nat
binNatGame' = Split iso binNatGame' binNatGame'
  where iso = Iso ask bld
    ask n = even n = Left (n ‘div’ 2)
    otherwise = Right (n ‘div’ 2)
    bld (Left m) = 2*m
    bld (Right m) = 2*m+1
```

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 \text{binNatGame'} and hence Lemma 4 cannot hold for this game.

### 3.5 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 bit counts.

For the the rest of this paper we embark in giving more ambitious and amusing concrete games for sets and \(\lambda\)-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 \text{type-directed} fashion. Indeed, we could even use \text{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 \text{listGame\ binNatGame} for this type. It would satisfy the basic round-tripping property (\text{Enc}/\text{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 would we encode the multiset \{3, 6, 5, 6\}? We might start by ordering the values to obtain the \text{canonical} representation \([3, 5, 6, 6]\). But now imagine encoding this using a vanilla list of natural numbers game \text{listGame\ binNatGame}: when encoding the second element, we would be wasting the codes for values 0, 1, and 2, as neither of these values could possibly follow 3 in the ordering. Instead of encoding the value 5 for the second element of the ordered list, we can encode 2, the \text{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 \text{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 decremented, taking account of the fact that the difference between successive elements must be non-zero.

In Haskell, we can implement \text{diff} and \text{undiff} functions that respectively compute and apply difference lists.

```haskell
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 multisets and sets of natural numbers, as follows.

```haskell
natMultisetGame :: Game Nat \rightarrow Game [Nat]
natMultisetGame g =
  listGame g \leftrightarrow Iso (diff (-) \circ\ sort) (undiff (+))
```
Here is the multiset game in action, using our binary encoding of natural numbers on the example multiset \{3, 6, 5, 6\}.

\[
\begin{align*}
> \text{enc (listGame binNatGame) [3,6,5,6]} & \Rightarrow [0,0,1,0,1,0,1,0,0,0,0,1,0,0,0,0,0,0,0,1,1] \\
> \text{enc (listGame binNatGame) [3,5,6,6]} & \Rightarrow [0,0,1,0,1,0,0,1,0,0,1,1,1] \\
\end{align*}
\]

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, 5, 6, 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 \(\lambda\)-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:

\[
\begin{align*}
\text{compareByGame :: Game a} & \rightarrow (a \rightarrow a \rightarrow \text{Ordering}) \\
\text{sortByGame :: Game a} & \rightarrow (a \rightarrow a) \\
\end{align*}
\]

\[
\begin{align*}
\text{compareByGame (Single x) y} & = \text{EQ} \\
\text{compareByGame (Split (Iso ask bld) g1 g2) x y} & = \\
& \begin{cases} 
\text{case (ask x, ask y) of} \\
(\text{Left x1, Left y1}) & \rightarrow \text{compareByGame g1 x1 y1} \\
(\text{Right x2, Right y2}) & \rightarrow \text{compareByGame g2 x2 y2} \\
(\text{Left x1, Right y2}) & \rightarrow \text{LT} \\
(\text{Right x2, Left y1}) & \rightarrow \text{GT} 
\end{cases} \\
\text{sortByGame :: Game a} & \rightarrow [a] \\
\text{sortByByGame g} & = \text{sortBy (compareByGame g)} \\
\end{align*}
\]

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 \text{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.

\[
\begin{align*}
\text{removeLE :: Game a} & \rightarrow a \rightarrow \text{Maybe (Game a)} \\
\text{removeLE (Single x)} x & = \text{Nothing} \\
\text{removeLE (Split (Iso ask bld) g1 g2) x} & = \\
& \begin{cases} 
\text{case ask x of} \\
\text{Left x1} & \rightarrow \text{case removeLE g1 x1 of} \\
\text{Nothing} & \rightarrow \text{Just (g2 \rightarrow right1)} \\
\text{Just g1’} & \rightarrow \text{Just (\text{Split (Iso ask bld) g1’ g2})} \\
\text{Right x2} & \rightarrow \text{case removeLE g2 x2 of} \\
\text{Nothing} & \rightarrow \text{Nothing} \\
\text{Just g2’} & \rightarrow \text{Just (g2’ \rightarrow right1)} \\
\end{cases} \\
\text{where right1} & = \text{Iso (\lambda y \rightarrow case ask x of Right y \rightarrow y) (bld \circ \text{Right})} \\
\end{align*}
\]

The code for listGame can then be adapted to do sets:

\[
\begin{align*}
\text{setGame :: Game a} & \rightarrow [a] \\
\text{setGame g} & = \text{setGame’ g} \Rightarrow \text{Iso (sortByByGame g) id} \\
& \begin{cases} 
\text{setGame’ g} & = \text{Split listIso unitGame $} \\
& \text{depGame g \& \lambda x \rightarrow} \\
& \text{case removeLE g x of} \\
& \text{Just g’} & \rightarrow \text{setGame’ g’} \\
& \text{Nothing} & \rightarrow \text{constGame []} \\
\end{cases} \\
\end{align*}
\]

Notice the dependent composition, which, once a value is determined, plays the game having removed all smaller elements from it.\(^6\)

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 \(\lambda\)-calculus, declared as a Haskell datatype using de Bruijn indexing for variables:

\[
\begin{align*}
\text{data Exp = Var Nat | Lam Exp | App Exp} \\
\end{align*}
\]

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

\[
\begin{align*}
\expGame :: \text{Nat} & \rightarrow \text{Game Exp} \\
\expGame 0 & = \text{appLamG 0} \\
\expGame n & = \begin{cases} 
\text{Split (Iso ask bld) (rangeGame 0 (n-1)) (appLamG n)} \\
\text{where ask (Var i)} & = \text{Left i} \\
\text{ask e} & = \text{Right e} \\
\text{bld (Left i)} & = \text{Var i} \\
\text{bld (Right e)} & = e 
\end{cases} \\
\end{align*}
\]

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

\[
\begin{align*}
\text{appLamG n} & = \\
& \text{Split (Iso ask bld) (prodGame (expGame n) (expGame n)) (expGame (n+1))} \\
& \begin{cases} 
\text{where ask (App e1 e2)} & = \text{Left (e1,e2)} \\
\text{ask (Lam e)} & = \text{Right e} \\
\text{bld (Left (e1,e2))} & = \text{App e1 e2} \\
\text{bld (Right e)} & = \text{Lam e} 
\end{cases} \\
\end{align*}
\]

For application terms we play \(\text{prodGame}\) 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’s run the game on the expression \(I K\) where \(I = \lambda x.x\) and \(K = \lambda x.\lambda y.x\).

\[
\begin{align*}
> \text{let tmI = Lam (Var 0)} \\
> \text{let tmK = Lam (Var 1)} \\
> \text{enc (expGame 0) (App tmI tmK)} \\
& \Rightarrow \text{dec (expGame 0) it} \\
& \Rightarrow \text{enc (expGame 0) (Lam (Var 0)) (Lam (Var 1)))},[] \\
\end{align*}
\]

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.

5.2 Simple types

We now move to the simply-typed \(\lambda\)-calculus, whose typing rules are shown in conventional form in Figure 5.

\(^6\) The \$ notation is just Haskell syntactic sugar that allows applications to be written with fewer parentheses: \(f \ (h \ g)\) can be written as \(f \ $ \ h \ g\).
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:

```haskell
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.

```haskell
typeOf :: Env → Exp → Ty

typeOf env (Lam t e) = TyArr t (typeOf (t:env) e)
typeOf env (App e _) = let TyArr _ t = typeOf env e in t
```

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 we 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], so we'd rather not go there (yet)!

We can make things easier for ourselves by solving a different problem: fix the type environment `env` and a type pattern of the form `1 → ... → 2` 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 returns a boolean indicating whether or not a type matches a pattern.

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

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

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

To define a game for typed terms 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.

```haskell
varGame :: (Ty → Bool) → Env → Maybe (Game Nat)
varGame f [] = Nothing
varGame f (t:env) = case varGame f t 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. Observe that 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.

```haskell
expGame :: Env → Pat → Game Exp

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) δ λ →
          expGame env (PArr (typeOf env e) p)
    lamG (PArr t p) = prodGame (constGame t) $ λ e →
                     expGame (t:env) p
    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 term 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 term is an application or a λ-abstraction. If the term can be a variable (case `Just varG`) then we may immediately `Split` with `varI` by asking if the term 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 term 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 pattern matches on the pattern argument. If it is an arrow pattern we play a dependent 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 a game for the argument type, bind the actual type to `t` and play `expGame` for the body of the abstraction using `t` to updated the environment.

That was it! Let's test `expGame` on the example expression from Section 1: `λx∶Nat.(x y)∶Nat.x`.

```haskell
> let ex = Lam TyNat (Lam TyNat (Var 1))
> enc (expGame [] Any) ex
[0,1,0,0,1,1,0]
> dec (expgame [] Any) it
```
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. Discussion

Non-proper filtering Sometimes it’s convenient not to be proper. Using voidGame from Section 3.4 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.

\[
\text{\textit{filterGame} \::\ (t \to \text{Bool}) \to \text{Game } t \to \text{Game } (t \to \text{Game } t)\\
\text{filterGame } f \circ g (\text{Single } (\text{Iso } \circ \text{bld})) = \begin{cases} \text{Split } (\text{Iso } \circ \text{bld}) & \text{if } f (\text{bld } \text{()) then } g \text{ else voidGame -- } (x:t \circ \text{Empty}!\\
\text{filterGame } f (\text{Split } (\text{Iso } \circ \text{bld}) \circ g) \text{g} \text{2} = \text{Split } (\text{Iso } \circ \text{bld}) (\text{filterGame } f \circ \text{bld } \circ \text{Left} \text{ g}1) (\text{filterGame } f \circ \text{bld } \circ \text{Right} \text{ g}2) \\
\end{cases}
\]

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.

\[
\text{dec } (\text{filterGame } \text{even } \circ \text{binNatGame}) \text{2} [1,1,0] \quad \text{dec } (\text{filterGame } \text{even } \circ \text{binNatGame}) [1,0,1,0,0,1,1,1,1]
\]

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

\[
\text{dec } (\text{filterGame } \text{even } \circ \text{binNatGame})[1,0,1,0,0,1,1,1,1] \quad \text{Exception: Input too short}
\]

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:

\[
\text{\textit{filterFinGame} } \::\ (t \to \text{Game } t) \to \text{Game } t \to \text{Maybe } (\text{Game } t)\\
\text{filterFinGame } f g (\text{Single } (\text{Iso } \circ \text{bld})) = \begin{cases} \text{filterGame } f (\text{Split } (\text{Iso } \circ \text{bld}) \circ g) & \text{if } f (\text{bld } \text{) then } g \text{ else voidGame}\\
\text{filterGame } f (\text{Split } (\text{Iso } \circ \text{bld}) \circ g) \text{2} = \text{filterGame } f \circ \text{bld } \circ \text{Left} \text{ g}1, \text{filterGame } f \circ \text{bld } \circ \text{Right} \text{ g}2) \text{of} (\text{Nothing}, \text{Nothing}) \to \text{Nothing}\\
\text{filterGame } f (\text{Split } (\text{Iso } \circ \text{bld}) \circ g) \text{2} = \text{filterGame } f \circ \text{bld } \circ \text{Left} \text{ g}1, \text{filterGame } f \circ \text{bld } \circ \text{Right} \text{ g}2) \text{of} (\text{Nothing}, \text{Nothing}) \to \text{Nothing}\\
\text{filterGame } f (\text{Split } (\text{Iso } \circ \text{bld}) \circ g) \text{2} = \text{filterGame } f \circ \text{bld } \circ \text{Left} \text{ g}1, \text{filterGame } f \circ \text{bld } \circ \text{Right} \text{ g}2) \text{of} (\text{Nothing}, \text{Nothing}) \to \text{Nothing}\\
\text{filterGame } f (\text{Split } (\text{Iso } \circ \text{bld}) \circ g) \text{2} = \text{filterGame } f \circ \text{bld } \circ \text{Left} \text{ g}1, \text{filterGame } f \circ \text{bld } \circ \text{Right} \text{ g}2) \text{of} (\text{Nothing}, \text{Nothing}) \to \text{Nothing}\\
\end{cases}
\]

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:

\[
\text{\textit{filterFinGame} } \::\ (t \to \text{Game } t) \to \text{Game } t \to \text{Maybe } (\text{Game } t)\\
\text{filterFinGame } f g (\text{Single } (\text{Iso } \circ \text{bld})) = \begin{cases} \text{filterGame } f (\text{Split } (\text{Iso } \circ \text{bld}) \circ g) & \text{if } f (\text{bld } \text{) then } g \text{ else voidGame}\\
\text{filterFinGame } f (\text{Split } (\text{Iso } \circ \text{bld}) \circ g) \text{2} = \text{filterGame } f \circ \text{bld } \circ \text{Left} \text{ g}1, \text{filterGame } f \circ \text{bld } \circ \text{Right} \text{ g}2) \text{of} (\text{Nothing}, \text{Nothing}) \to \text{Nothing}\\
\text{filterGame } f (\text{Split } (\text{Iso } \circ \text{bld}) \circ g) \text{2} = \text{filterGame } f \circ \text{bld } \circ \text{Left} \text{ g}1, \text{filterGame } f \circ \text{bld } \circ \text{Right} \text{ g}2) \text{of} (\text{Nothing}, \text{Nothing}) \to \text{Nothing}\\
\text{filterGame } f (\text{Split } (\text{Iso } \circ \text{bld}) \circ g) \text{2} = \text{filterGame } f \circ \text{bld } \circ \text{Left} \text{ g}1, \text{filterGame } f \circ \text{bld } \circ \text{Right} \text{ g}2) \text{of} (\text{Nothing}, \text{Nothing}) \to \text{Nothing}\\
\end{cases}
\]

Since we do not have expGame in Coq, we’ve only showed this on paper, hence it’s a Proposition and not a Theorem.
we navigate down the tree, pointers to thunks representing
Determining the space complexity of games is somewhat tricky: as
an every-bit-counts codec for ML-style let polymorphism.)

such as Haskell Core [23] or .NET CIL [7]. (We've already created
the technique to a reasonably-sized compiler intermediate language
approach is suitable only for theoretical investigations but not for
There is no reason to believe that the game-based
inside

pointers is relevant. An optimization would involve embedding the
left and the right subtrees are kept around, although only one of two

obtain a proper game for simply-typed terms!

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 filterGame_inf: perform a
breadth-first traversal of the original game, and each time you en-
counter 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 terms of a given type in a given environ-
ment, from the weaker games that expGameCheck of Section 5.3
produces, if we have a precondition that there exists one term of
the given type in the given environment. If there exists one term 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 terms!

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
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:

Compression For reasons of space, we have compressed away
different 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 statistics in our games
for typed programs [3], producing codes that are even more compact
than is attained through the use of type information.

Games for test generation Test generation for use in tools such as
Quickcheck [4] is a potential application of game-based decoding,
since generating random bitstrings amounts to generating pro-
grams. 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.

Program development and verification in Coq Our attempts to
encode everything in this paper in Coq stumbled upon Coq's lim-
ited support for co-recursion, namely the requirement that recur-
sive 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 gener-
ation facilities of Progol [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), and
the isomorphism obligations could be proven on the side. Our over-
all conclusion from the experience is that Coq itself can become a
very effective development platform but it would benefit from bet-
ter support for general recursion, co-recursion, and type inference.

7. Related work

Our work has strong connections to Kennedy's pickler combina-
tors [16]. There, a codec was represented by a pair of encoder and
decoder functions, with codecs for complex types built from simple
one using combinators. The basic round-trip property (Enc/Dec)
was considered informally, but stronger properties were not stud-
ted. 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 the encoder and decoder out
of necessity used the same 'logic'. In our opinion, games are more
succinct 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 proper-
ties for encoders and decoders in several encryption schemes.

> enc (fromJust (filterFinGame even (rangeGame 0 7))) 4
[1,0]

Compare this to the original encoding before filtering:

> enc (rangeGame 0 7) 4
[1,0,0]

Being non-proper can also be useful for padding [17] to create a
fixed length code, or one which is a multiple of some block size.
In our setting this is easy to do; here's a straightforward padding
strategy: in singleton leaves whose path from the root is smaller
than the desired length append a game that splits that singleton set
into the empty set and the original singleton set. Play voidGame for
the empty set, and repeat until you have reached the desired path
length where you may insert the original singleton node.

}
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. There exist many approaches to datatype-generic programming that can address marshalling and unmarshalling [14]. Most of these approaches 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 are data description languages, which associate the semantics of types to their low-level representations [9]. The interpretation 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 been 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 proverness property expresses, which provably leads to every bit only asking questions that ‘make sense’. That is precisely what our tamper-proof) codes for low-level bytecode [13, 10]. Although that claim must be made on an admissible basis 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.

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.

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.

References


