

# **Verificação formal de uma implementação eficiente de um decodificador de UTF-8**

**Leonardo Santiago**  
leonardors@dcc.ufrj.br  
UFRJ

## ABSTRACT

O sistema de codificação *Unicode* é imprescindível para a comunicação global, permitindo que inúmeras linguagens utilizem a mesma representação para transmitir todas os caracteres, eliminando a necessidade de conversão. Dentre todos os formatos de serializar caracteres do Unicode - denominados *codepoints* - certamente o formato mais ubíquo é o UTF-8, pela sua retro compatibilidade com ASCII, e a capacidade de economizar bytes. Apesar de ser utilizado em mais de 98% das páginas da internet, vários problemas aparecem ao implementar programas de codificação e decodificações de UTF-8 semanticamente corretos, e inúmeras vulnerabilidades estão associadas a esse processo. Dificultando ainda mais, a especificação dada pelo Consórcio Unicode é feita inteiramente em prosa, tornando extremamente difícil afirmar com segurança que dada implementação respeita-a por métodos tradicionais. Assim, este trabalho utilizará verificação formal através de provadores de teoremas interativos de duas formas: primeiro, será desenvolvido um conjunto de propriedades - a especificação - que unicamente representam um par de programas codificador e decodificador de UTF-8. Com a especificação formalizada, serão implementados um codificador e decodificador, mostrando que esses respeitam todas as propriedades necessárias para que estejam corretos.

## Sumário

1	Introdução .....	1
2	Unicode .....	2
2.1	UCS-2 e UTF-16 .....	3
2.2	UTF-8 .....	5
3	Revisão de literatura .....	7
3.1	Trabalhos relacionados .....	8
4	Formalização da especificação .....	9
4.1	Corretude da especificação .....	15
4.2	Ordenações em conjuntos finitos .....	16
4.3	Índices de codepoints e de sequências de bytes .....	18
4.4	Especificação implica mapeamento UTF-8 correto .....	22
5	Implementação .....	26
5.1	Corretude da implementação .....	30
	Bibliografia .....	30

## 1 Introdução

O processo de desenvolvimento de software pode ser separado em duas fases distintas: a de validação, que pretende desenvolver especificações necessárias para que um programa resolva um problema no mundo real, e a de verificação, que assegura que o programa desenvolvido implementa essas especificações.

Especificação é o principal tópico de estudo das práticas de modelagem de software, que tem como produção gráficos conceituais, modelos e regras de negócio, que devem ser utilizados para desenvolver o programa. O objetivo dessas é gerar um conjunto de objetivos e propriedades que programas devem satisfazer para que atinjam algum fim no mundo real, conferindo semântica à resultados e implementações, e construindo pontes tangíveis entre modelos teóricos e a realidade prática.

Assegurar que dada implementação segue as regras de negócio geradas na fase de especificação é tópico de estudo da área de verificação. Dela, inúmeras práticas comuns na área de programação são derivadas, como desenvolvimento de testes, garantias de qualidade e checagens de tipo. Apesar das inúmeras práticas, preencher a lacuna entre a semântica dos modelos teóricos e as implementações em código é extremamente difícil, dada a natureza das práticas tradicionais baseadas em testes unitários. Testes oferecem visões circunstanciais do comportamento do programa a partir de certas condições iniciais, tornando impossível assegurar com totalidade a corretude do programa, visto que programas complexos teriam de ter um número impraticável de testes – muitas vezes infinito – para checar todas as combinações de condições iniciais.

É cotidiano que erros passem despercebidos por baterias gigantescas de testes e apareçam somente em produção – quando erros são inaceitáveis – em especial quando ocorrem em combinações muito específicas de entrada. Muitas linguagens então tomam uma abordagem dinâmica, isto é, tornar erros mais fáceis de serem detectados adicionando inúmeras checagens enquanto o programa executa, e tornando-o programa ainda mais fácil de quebrar. Para atingir *software* correto, é imprescindível a análise estática dos programas, mas técnicas comuns de análise estática não são potentes o suficiente para conferir segurança e corretude, e são significativamente mais complexas do que abordagens dinâmicas.

Verificação formal de software denomina a área da verificação que oferece diretrizes para raciocinar formalmente sobre um programa, descrevendo axiomas, regras e práticas que permitem construir provas sobre o comportamento desse. Ao estruturar o programa para permitir o raciocínio matemático, torna-se possível atribuir uma semântica a um software, conferindo fortes garantias de corretude, e assegurando-se que esse está conforme as especificações da semântica. Para auxiliar nesse processo, várias ferramentas foram desenvolvidas, como *model checkers*, que tentam gerar provas automaticamente a partir de modelos fornecidos, e provadores de teorema interativos, que permitem o desenvolvedor elaborar provas sobre programas utilizando linguagens específicas para construí-las.

Por necessitar que programas sejam estruturados de maneira a facilitar o raciocínio lógico, a metodologia da verificação formal dificilmente é aplicada a projetos complexos já existentes, visto que tradicionalmente são feitos com outros objetivos em mente – facilidade de desenvolvimento, agilidade em desenvolver novas capacidades, ou até mesmo velocidade do programa gerado. Além disso, as ferramentas mais poderosas de verificação formal, os provadores de teoremas interativos, utilizam tipos dependentes, que nativamente utilizam linguagens funcionais para sua lógica interna, o que significa que expressar programas imperativos nessas geralmente

requer muito mais trabalho. Assim, fica claro que existem certas barreiras para a adoção de métodos formais na indústria.

O objetivo deste trabalho é, portanto, documentar os benefícios, bem como as dificuldades, da aplicação desses métodos a problemas suficientemente complexos, de forma a confirmar ou refutar o estigma existente na adoção da verificação formal. Em particular, o problema da codificação e decodificação de caracteres em UTF-8 foi escolhido pela sua difusão em praticamente todos os contextos e linguagens de programação.

O padrão Unicode (THE UNICODE CONSORTIUM (2025)) de representação de caracteres é ubíquo na comunicação na internet, e seu principal formato de codificação e decodificação, UTF-8, é utilizado em mais de 98% das páginas web (W3TECHS (2025)). Apesar disso, inúmeras CVEs estão associadas a programas que tratam UTF-8 incorretamente, especialmente por não implementarem totalmente a especificação, visto que muitos casos incomuns podem acabar sendo esquecidos.

As vulnerabilidades CVE-2000-0884 (Microsoft IIS) e CVE-2008-2938 (APACHE Tomcat) estão diretamente associadas à má gestão de input ao ler caracteres UTF-8, permitindo ao atacante de ler arquivos em caminhos fora do inicialmente permitido (ataque conhecido como *directory traversal*). A CVE-2004-2579 (Novell iChain) está associada a um ataque que utiliza representações ilegais de caracteres de escape em UTF-8 para ultrapassar regras de controle. Além disso, o leitor de UTF-8 da linguagem PHP em versões mais antigas não tratava corretamente casos especiais desse sistema, tornando possível injeções de SQL (CVE-2009-4142), *cross site scripting* (CVE-2010-3870), e *integer overflows* (CVE-2009-5016). Dessa forma, fica claro que a formalização formal como forma de assegurar corretude e segurança é uma ferramenta valiosa.

Este trabalho é estruturado nas seguintes seções:

1. Na seção 2, a história por trás do sistema Unicode será revista, com o objetivo de motivar a estruturação atual dos sistemas de codificação UTF-8, UTF-16 e UTF-32, bem como algumas de suas propriedades e limitações.
2. Na seção 3, será inspecionada a literatura existente, tanto especificações existentes do Unicode quanto sobre abordagens e metodologias tradicionais de provar formalmente a corretude de codificadores e decodificadores de linguagens.
3. Na seção 4, será elaborado um conjunto de regras formais que um codificador e decodificador, denominado de **especificação**, e serão provados teoremas que fundamentam a corretude desse.
4. Na seção 5, serão desenvolvidos implementações práticas de um codificador e decodificador UTF-8, levando em consideração fatores como simplicidade, utilidade e eficiência, de maneira similar a como são implementados em linguagens “imperativas”.
5. Na seção 6, serão dadas as considerações finais, bem como aplicações naturais desse trabalho para cenários práticos.

(Deveria citar que a seção 4 é “inovadora”? no sentido de que não existem trabalhos que fazem isso hoje em dia.)

## 2 Unicode

Sistemas de codificação são padrões criados para transformar caracteres em números, como A=65, Ã=195 e 語=35486, e posteriormente serializá-los em mensagens para enviá-los a outras pessoas. O padrão Unicode é o sistema de representação de caracteres mais utilizado mundialmente hoje em dia, por objetivar incluir todas as linguagens existentes de maneira integrado.

O padrão define 3 esquemas de codificação distintos para transformar caracteres Unicode em seqüências de bits: UTF-8, UTF-16 e UTF-32. Para entender o design e funcionamento desses, faz-se necessário entender como funcionavam os antecessores.

Definição: **code point** (ou **valor escalar**) é o nome dado à representação numérica de um caractere. No formato Unicode, é comum representá-los no formato **U+ABCDEF**, onde **ABCDEF** armazena o número do *code point* em hexadecimal.

Definição: um **codificador** é um programa que recebe valores escalares e transforma-os em seqüências de bits, e um **decodificador** é um programa que lê seqüências de bits e transforma-os de volta em valores escalares.

Sem dúvidas o sistema de codificação mais influente da história é o ASCII. Criado para servir as necessidades da indústria americana de *teleprinters*, o ASCII define apenas 127 caracteres, focando principalmente em compactar a quantidade de bits necessários para enviar uma mensagem, de forma que todo caractere pode ser expresso utilizando apenas 7 bits.

Com a evolução dos computadores, e a consolidação de um byte como 8 bits, muitos sistemas de codificação surgiram mantendo os primeiros 127 caracteres iguais a ASCII, e adicionando 128 caracteres no final, utilizando o oitavo bit previamente ignorado. Esses foram criados primariamente para adicionar suporte a caracteres específicos de cada linguagem, como ã, ç, e €, de modo a manter compatibilidade com o ASCII, e ficaram conhecidos como codificações de ASCII estendido.

Tanto o ASCII quanto suas extensões utilizam um mapeamento um pra um entre o número dos caracteres e os bits das suas representações, tanto por simplicidade de codificação quanto por eficiência de armazenamento de memória. Programas que decodificam bytes em caracteres nesses sistemas são extremamente simples, e podem ser resumidos a tabelas de conversão direta, conhecidas como *code pages*.

Apesar da simplicidade dos programas, representar um byte por caractere coloca uma severa limitação no número de caracteres que conseguem expressar ( $\leq 256$ ), fazendo com que cada linguagem diferente tivesse sua própria maneira distinta de representar seus caracteres, e que muitas vezes era incompatível com as outras. Assim, enviar textos pela internet era uma tarefa complicada, visto que não era garantido que o usuário que recebe a mensagem teria as tabelas necessárias para decodificá-la corretamente.

Para piorar a situação, linguagens baseadas em ideogramas, como japonês, coreano e chinês possuem milhares de caracteres, e codificá-las em apenas um byte é impossível. Tais linguagens foram pioneiras em encodings multi-bytes, em que um caractere é transformado em mais de um byte, tornando a codificação e decodificação significativamente mais complexa.

O padrão Unicode foi criado então para que um único sistema de codificação consiga cobrir todas as linguagens, com todos seus caracteres específicos, de forma que qualquer texto escrito em qualquer linguagem possa ser escrito nele. Apesar de ambicioso, esse sistema rapidamente ganhou adoção mundial, simplificando a comunicação na internet.

## 2.1 UCS-2 e UTF-16

Em 1991, a versão 1.0 do Unicode foi lançado pelo consórcio Unicode, com uma codificação de tamanho fixo de 16 bits conhecida por UCS-2 – *Universal Coding System* – capaz de representar

65536 caracteres das mais diversas línguas. Rapidamente, esse sistema ganhou adoção em sistemas de grande relevância, como o sistema de UI Qt (1992), Windows NT 3.1 (1993) e até mesmo linguagens como Java (1995).

Tal quantidade, apesar de muito maior do que os antigos 256, rapidamente provou-se não suficiente para todas as linguagens. Quando isso fora percebido, o sistema UCS-2 já estava em amplo uso, e trocá-lo por outro sistema já não era mais uma tarefa trivial. Assim, para estendê-lo mantendo-o retro compatível, decidiram reservar parte da tabela de caracteres para que dois caracteres distintos (32 bits) representem um único *code point*. Dessa forma, o sistema deixou de ter um tamanho fixo de 16 bits, e passou a ter um tamanho variável, dependendo de quais *code points* são codificados.

O padrão UCS-2 estendido com *surrogate pairs* tornou-se oficialmente o padrão UTF-16 (*Unicode Translation Format*) na versão 2.0 do Unicode. Desde então, o uso do UCS-2 é desencorajado, visto que UTF-16 é considerado uma extensão em todos os aspectos a ele. Hoje em dia, na versão 17.0 do padrão Unicode, 297,334 *code points* já foram definidos, muito além da projeção inicial de 65536.

Para determinar se uma sequência de bytes é válida em UTF-16, faz-se necessário determinar se o primeiro byte representa o início de um *surrogate pair*, representado por bytes entre D800 e DBFF, seguido de bytes que representam o fim de um *surrogate pair*, entre DC00 e DFFF. O esquema de serialização pode ser visto da seguinte forma:

Início..Fim	Bytes	Bits relevantes
U+0000 U+FFFF	wwwxxxxx yyyyzzzz	16 bits
U+10000 U+10FFFF	110110vv vvwwwwxx 110111xx yyyyzzzz	20 bits

Tabela 1: Distribuição dos bits em bytes válidos UTF-16.

Assim, para que a decodificação de UTF-16 seja não ambígua, é necessário que *code points* do primeiro intervalo, que não possuem cabeçalho para diferenciá-los, não possam começar com a sequência de bits 11011. Além disso, iniciar um *surrogate pair* (D800..DBFF) e não terminá-lo com um *code point* no intervalo correto (DC00..DFFF) é considerado um erro, e é inválido segundo a especificação. De fato, o padrão Unicode explicita que **nenhum** *code point* pode ser representado pelo intervalo U+D800..U+DFFF, de forma que todos os outros sistemas de codificação – UTF-8, UTF-32 – tenham que desenvolver sistemas para evitar que esses sejam considerados *code points* válidos.

A quantidade de *code points* definidos pelo Unicode está diretamente ligada à essas limitações do padrão UTF-16, que consegue expressar 1.112.064 *code points*. Esse número pode ser calculado da seguinte forma:

Início..Fim	Tamanho	Descrição
U+0000..U+FFFF	$2^{16}$	Basic Multilingual Plane, Plane 0
U+D800..U+DFFF	$2^{11}$	Surrogate Pairs
U+10000..U+10FFFF	$2^{20}$	Higher Planes, Planes 1-16
U+0000..U+10FFFF \ U+D800..U+DFFF	$2^{20} + 2^{16} - 2^{11}$	<i>Code points</i> representáveis

Tabela 2: Intervalos de *code points* válidos.

Disso, pode-se inferir que um *code point* **válido** é um número de 21 bits que:

1. Não está no intervalo U+D800..U+DFFF.

## 2. Não ultrapassa U+10FFFF.

Vale notar que há ambiguidade na forma de serializar UTF-16 para bytes, visto que não é especificado se o primeiro byte de um *code point* deve ser o mais significativo – Big Endian – ou o menos significativo – Little Endian. Para distinguir, é comum o uso do caractere U+FEFF, conhecido como *Byte Order Mark* (BOM), como o primeiro caractere de uma mensagem ou arquivo. No caso de Big Endian, o BOM aparece como FEFF, e no caso de Little Endian, aparece como FFFE.

Essa distinção é o que faz com que UTF-16 possa ser dividido em duas sub linguagens, UTF-16BE (Big Endian) e UTF-16LE (Little Endian), adicionando ainda mais complexidade à tarefa de codificar e decodificar os caracteres corretamente.

Com essas complexidades, implementar codificação e decodificação de UTF-16 corretamente tornou-se muito mais complicado. Determinar se uma sequência de bytes deixou de ser uma tarefa trivial, e tornou-se um possível lugar onde erros de segurança podem acontecer. De fato, CVE-2008-2938 e CVE-2012-2135 são exemplos de vulnerabilidades encontradas em funções relacionadas à decodificação em UTF-16, em projetos grandes e bem estabelecidas (APACHE e Python, respectivamente).

Apesar de extremamente útil, o UTF-16 utiliza 2 bytes para cada caractere, então não é eficiente para linguagens cujos caracteres encontram-se no intervalo original do ASCII (1 byte por caractere), como os formatos HTML e JSON utilizados na internet, que usam muitos caracteres de pontuação – <, >, {, :. Por isso, fez-se necessário achar outra forma de codificá-los que fosse mais eficiente para a comunicação digital.

## 2.2 UTF-8

Criado por Rob Pike e Ken Thompson, o UTF-8 surgiu como uma alternativa ao UTF-16 que utiliza menos bytes. A principal mudança para que isso fosse possível foi a de abandonar a ideia de codificação de tamanho fixo desde o início, que imensamente facilita escrever os programas decodificadores, preferindo uma codificação de tamanho variável e utilizando cabeçalhos em todos os bytes para evitar que haja ambiguidade.

A quantidade de bytes necessários para representar um *code point* em UTF-8 é uma função do intervalo que esse *code point* se encontra. Ao invés de serializar os *code points* diretamente, como o UTF-16 fazia, agora todos os bytes contém cabeçalhos, que indicam o tamanho da serialização do *code point* – isto é, a quantidade de bytes a seguir.

Para *code points* no intervalo U+0000..U+007F, apenas 1 byte é usado, e esse deve começar com o bit 0. Para *code points* no intervalo U+0080..07FF, dois bytes são usados, o primeiro começando com os bits 110, e o segundo sendo um byte de continuação, que contém o cabeçalho 10. Para aqueles no intervalo U+0800..U+FFFF, o primeiro byte deve começar com 1110, seguido de dois bytes de continuação, e por fim, aqueles no intervalo U+10000..U+10FFFF, o primeiro byte deve começar com 11110, seguido de três bytes de continuação.

Considerando que um *code point* precisa de 21 bits para ser armazenado, podemos separar seus bits como [u, vvvv, www, xxxx, yyyy, zzzz]. Utilizando essa notação, a serialização deste pode ser vista como:



Início..Fim	Byte 1	Byte 2	Byte 3	Byte 4	Bits relevantes
U+0000 U+007F	0yyyzzzz				7 bits
U+0080 U+07FF	110xxxxy	10yyzzzz			11 bits
U+0800 U+FFFF	1110www	10xxxxy	10yyzzzz		16 bits
U+10000 U+10FFFF	11110uvv	10vvwww	10xxxxy	10yyzzzz	21 bits

Tabela 3: Distribuição dos bits em bytes UTF-8.

É importante notar que os primeiros 127 *code points* são representados exatamente igual caracteres ASCII (e sistemas estendidos), algo extremamente desejável não apenas para compatibilidade com sistemas antigos, mas para recuperar parte da eficiência de espaço perdida no UTF-16. Diferentemente do UTF-16, o UTF-8 também não possui ambiguidade de *endianness*, e portanto não precisa utilizar o BOM para distinguir; há apenas uma maneira de ordenar os bytes.

O UTF-8 ainda precisa manter as limitações do UTF-16. Como *surrogate pairs* não são mais utilizados para representar *code points* estendidos, é necessário garantir que bytes do intervalo D800..DFFF não apareçam, já que não possuem significado.

Além disso, apesar de conseguir codificar 21 bits no caso com maior capacidade (U+0000..U+10FFFF), nem todos desses representam *code points* válidos, visto que o padrão Unicode define-os baseando nos limites do UTF-16. Isso significa que o codificador deve assegurar de que todos *code points* decodificados não sejam maior do que U+10FFFF.

As primeiras versões da especificação do UTF-8 não faziam distinção de qual o tamanho deveria ser utilizado para codificar um *code point*. Por exemplo, o caractere A é representado por U+0041 = 1000001. Isso significa que ele podia ser representado em UTF-8 como qualquer uma das seguintes sequências:

Sequência de bits	Hexadecimal
01000001	41
11000001 10000001	C1 81
11100000 10000001 10000001	E0 81 81
11110000 10000000 10000001 10000001	F0 80 81 81

Tabela 4: Possíveis representações para o caractere U+0041.

Permitir tais codificações causou inúmeras vulnerabilidades de segurança, visto que vários programas erroneamente ignoram a noção de *code points* e tratam esses como sequências de bytes diretamente. Ao tentar proibir certos caracteres de aparecerem em uma string, os programas procuravam por sequências de bytes especificamente, ao invés de *code points*, e ignoravam que um *code point* podia ser codificado de outra forma. Várias CVEs estão ligadas diretamente à má gestão dessas possíveis formas de codificar *code points* (desenvolver mais).

O padrão Unicode nomeou esses casos como *overlong encodings*, e modificou especificações futuras para que a única codificação válida de um *code point* em UTF-8 seja a menor possível. Isso adiciona ainda mais dificuldade na hora de decodificar os bytes, visto que o conteúdo do *code point* deve ser observado, para checar se fora codificado do tamanho certo.

Assim, validar que uma sequência de bytes representa UTF-8 válido significa respeitar as seguintes propriedades:

1. Nenhum byte está no intervalo de *code points* de *surrogate pairs* (U+D800..U+DFFF), e consequentemente, nenhum *code point* deve ocupar esse intervalo também.
2. Todo *code point* lido é menor ou igual a U+10FFFF
3. Todo *code point* é escrito na menor quantidade de bytes necessária para expressá-lo, isto é, não há *overlong encoding*.
4. Todo byte de início começa com o header correto (a depender do intervalo do *codepoint*).
5. Todo byte de continuação começa com o header correto (10).

### 3 Revisão de literatura

A proposição original do sistema de codificação UTF-8 fora dada no RFC3629, que passou por múltiplas revisões, até ser oficialmente transferida para a especificação Unicode, a partir de sua versão 4.0, em 2003. Desde então, a definição autoritária para esse esquema é dada pelo Consórcio Unicode, dentro da especificação geral do sistema Unicode (citação?).

No capítulo 3.9 da especificação do sistema Unicode, são definidos conceitos gerais de codificação, bem como os formatos UTF-8, UTF-16 e UTF-32. Nesse capítulo, duas definições importantes são feitas:

1. [D77] **Valor escalar:** um valor escalar Unicode é qualquer code point que não está no intervalo de *surrogate pairs*. Essa definição é a mesma de code points válidos dada anteriormente.
2. [D79] **Esquema de codificação Unicode:** um mapeamento único entre um valor escalar e uma sequência de bytes. A especificação oferece a definição de três esquemas de codificação oficiais: UTF-32 ([D90]), UTF-16 ([D91]) e UTF-8 ([D92]).

Segundo a definição D92, o UTF-8 é um formato de codificação que transforma um escalar Unicode em uma sequência de 1 a 4 bytes, cujos bits representam code points assim como especificado na Tabela 3. Para decidir quais bytes são válidos, é oferecida a tabela 3.7, reproduzida abaixo em verbatim:

Início..Fim	Byte 1	Byte 2	Byte 3	Byte 4
U+0000 U+007F	00..7F			
U+0080 U+07FF	C2..DF	80..BF		
U+0800 U+0FFF	E0	A0..BF	80..BF	
U+1000 U+CFFF	E1..EC	80..BF	80..BF	
U+D000 U+D7FF	ED	80..9F	80..BF	
U+E000 U+FFFF	EE..EF	80..BF	80..BF	
U+10000 U+3FFFF	F0	90..BF	80..BF	80..BF
U+40000 U+FFFFF	F1..F3	80..BF	80..BF	80..BF
U+100000 U+10FFFF	F4	80..8F	80..BF	80..BF

Tabela 5: Sequências de bytes UTF-8 bem formadas.

Os intervalos 80..BF representam os intervalos comuns de continuação – isto é, bytes que começam com 10 sempre estão nesse intervalo – e portanto, os bytes que diferem desses estão marcados em negrito. Essas diferenças são necessárias para evitar os casos de *overlong encoding* – onde o *code point* representado caberia em uma representação menor – e de *surrogate pair* – onde o *code point* representado estaria no intervalo D800..DFFF.

No caso em que o *code point* está no intervalo ASCII, ele é representado sem restrições. Quando é necessário dois bytes, o primeiro não pode começar com C0 ou C1 pois faria o *code*

*point* resultante caber no intervalo anterior. No caso de 3 bytes, há a possibilidade de o *code point* equivalente estar no intervalo D800..DFFF, e por isso é separado em 4 intervalos distintos. O primeiro intervalo se preocupa em impedir que ocorra *overlong encoding*, restringindo o segundo byte; o segundo intervalo contém apenas bytes estritamente menores do que U+D000; o terceiro intervalo restringe o segundo byte para garantir que seja menor do que U+D7FF; o último intervalo representa aqueles estritamente maiores do que U+DFFF. Da mesma forma, o caso de 4 bytes é separado em três. O primeiro caso se preocupa em impedir *overlong encoding*, enquanto o último caso garante que o *code point* seja estritamente menor do que U+10FFFF (o maior *code point* válido).

O problema com essa especificação é a falta de clareza entre a tabela descritiva e as propriedades intrínsecas ao UTF-8. Não é óbvio que há uma correspondência única entre sequências de bytes e *code points* válidos, nem que todo *code point* representado por esse formato é necessariamente válido. Além disso, as operações de extração e concatenação de bits oferecidos pela Tabela 3 não são triviais, e são suscetíveis a erros. Com uma especificação complicada demais, é possível que erros sejam cometidos até mesmo na concepção das regras. Quanto menor o conjunto de regras, mais fácil é de conferir manualmente que elas estão corretas.

### 3.1 Trabalhos relacionados

Faz-se necessário, portanto, estudar como codificadores e decodificadores são especificados e formalizados tradicionalmente na academia. Em geral, para mostrar a **corretude funcional** de ambos, é interessante mostrar que o codificador e decodificador recuperam os valores de entrada originais um do outro. Isto é, a grosso modo, mostrar que  $\text{encoder } a = b$  se, e somente,  $\text{decoder } b = a$ .

YE; DELAWARE (2019) descrevem o processo de implementar em Rocq um gerador de codificador e decodificador para Protobuf. Como o protocolo permite que o usuário gere formatos binários baseado em arquivos de configuração, os autores oferecem uma formalização da semântica para os arquivos *protocol buffers*, e utilizam-a para gerar programas que codificam e decodificam os formatos especificados em um arquivo, junto das provas de que os programas gerados devem obedecer a essa semântica corretamente e que esses necessariamente são inversos um do outro.

KOPROWSKI; BINSZTOK (2010) forneceram uma implementação similar para linguagens que podem ser descritas por PEGs em Rocq, junto de exemplos práticos de implementações de parsers de XML e da linguagem Java. GEEST; SWIERSTRA (2017) desenvolveram uma biblioteca em Agda para descrever pacotes em formários abstratos, focando no caso de uso dos padrões ASN.1, fornecendo uma formalização de formato IPV4. Ambos utilizam a noção de inversibilidade entre o codificador e decodificador como fundamento para a corretude.

THÉRY (2004) formalizou uma implementação do algoritmo de Huffman, frequentemente utilizado em padrões de compressão sem perda de dados. Similarmente SENJAK; HOFMANN (2016) construíram uma implementação completa do algoritmo de Deflate, usado em formatos como PNG e GZIP. Para mostrar a corretude, ambos provam a corretude mostrando que o codificador e decodificador são inversos.

DELAWARE et al. (2019) desenvolveram uma biblioteca em Rocq, *Narcissus*, que permite o usuário de descrever formatos binários de mensagens em uma DSL dentro do provador interativo. A principal contribuição do artigo é utilizar o maquinário nativo de Rocq para derivar tanto as implementações e as provas utilizando macros de forma que o sistema seja extremamente expressivo. Em casos que a biblioteca não é forte o suficiente para gerar as provas,

o usuário é capaz de fornecer provas manualmente escritas para a corretude, de forma a estender as capacidades do sistema.

RAMANANANDRO et al. (2025) desenvolveram uma biblioteca parecida chamada *PulseParse* na linguagem F\*, para implementar serializadores e desserializadores para vários formatos: CBOR, um formato binário inspirado em JSON, e CDDL, uma linguagem que especifica formatos estáticos CBOR. Utilizando essa biblioteca, os autores fornecem uma semântica ao CDDL e provam a corretude de programas gerados em cima desse conforme essa semântica.

Para a simplicidade de implementação, a formalização dada neste trabalho não utilizará nenhuma biblioteca, visto que essas introduzem complexidades específicas de cada DSL. Assim, quase tudo será feito do zero.

## 4 Formalização da especificação

Visto que a especificação fornecida pelo Consórcio Unicode não é forte o suficiente, tornou-se necessário estabelecer precisamente quais as propriedades que o codificador e decodificador devem satisfazer pra que sejam considerados corretos. Como visto nos outros trabalhos, é interessante conseguir provar que quaisquer codificador e decodificador que respeitam a especificação devem necessariamente ser inversos um do outro, entretanto isso não é suficiente, visto que é possível que a especificação contenha algum erro, e que não represente exatamente o mapeamento correto, mas ainda sim faça com que as funções sejam inversas.

Conceitualmente, há duas preocupações em formalizar um codificador e decodificador para garantir a corretude da especificação. A primeira é como identificar unicamente o mapa entre codepoints e sequências de bytes, e a segunda é como representar sequências de codepoints e sequências de bytes, de forma que seja possível aplicar o mapa anterior repetidamente, acumulando seu resultado.

Para representar tanto *code points* quanto *bytes* será utilizado o tipo `Z`, que representa o conjunto dos inteiros em Coq, pois ele possui uma grande gama de provas e propriedades já feitas anteriormente, de modo que muitas relações matemáticas possam ser reutilizadas. Quanto a segunda preocupação, em linguagens funcionais, é tradicional representar strings como listas encadeadas, de forma que tanto as sequências de bytes quanto sequências de codepoints sejam representados como listas encadeadas de números:

```
Definition codepoint : Type := Z.
Definition byte : Type := Z.

Definition unicode_str : Type := list codepoint.
Definition byte_str : Type := list byte.
```

Assim, faz sentido considerar que ambos o codificador e o decodificador sejam funções que mapeiam uma lista de números em uma nova lista de números, mas isso não leva em consideração que ambas podem receber argumentos inválidos. De fato, é necessária uma maneira de sinalizar que a lista retornada não era uma sequência UTF-8 válida.

Para formalizar codificadores e decodificadores, será utilizada a noção de *parser*. De modo geral, *parsers* processam elementos de tipo `A` e retornam algum valor de tipo `B`, quando a transformação pode não funcionar em todos os casos. Assim, é tradicional utilizar alguma estrutura que envolve o resultado `B` em múltiplos casos para representar a falibilidade.

O exemplo mais comum dessa estrutura é `option B`, que pode ser tanto `Some` com um valor de tipo `B`, ou `None`, representando que o *parser* falhou em extrair informação da entrada.

```

Inductive option (B :Type) : Type :=
| Some : B -> option B
| None : option B.

```

Entretanto, o problema de utilizar o tipo `option` é que é possível que uma sequência de bytes seja quase inteiramente UTF-8 válida, mas tenha algum erro por corrupção na hora da transmissão. Nesse caso, o `parser` retornaria `None`, e toda informação seria descartada. Ao invés disso, é útil exigir que o `parser` tente sempre ler o maior número de bytes o possível do prefixo da entrada, e ao encontrar bytes inválidos, substitua-os pelo caractere ‘`◆`’ (U+FFFD). Essa prática é tão difundida que o capítulo 3.9.6 do padrão Unicode dá guias gerais sobre como essa substituição deve ser feita.

Este trabalho é restringido à leitura de prefixo válido na entrada, pois especificar o algoritmo de substituição pode ser feito em um trabalho futuro, como um *parser* que roda o decodificador UTF-8 e substitui as partes inválidas de acordo com o que especificado no capítulo 3.9.6.

Assim, um *parser* parcial é definido como uma função que recebe uma lista de elementos de tipo `input` e retorna um par de `output` e lista de `input`. A semântica de um *parser* parcial é que a lista de `output` representa o resultado de “consumir” o prefixo válido da lista de entrada, enquanto a lista de `input` no resultado representa o sufixo não consumido. Essa semântica é enforçada como propriedades na especificação, vistas mais a frente.

```

Definition partial_parser (input output: Type) := list input -> (output * list
input).

```

```

Definition encoder_type := partial_parser codepoint (list byte).

```

```

Definition decoder_type := partial_parser byte (list codepoint).

```

Para especificar unicamente o mapeamento entre sequências de bytes e codepoints, devem ser utilizadas as tabelas Tabela 3 e Tabela 5. Uma possível maneira de traduzir isso em código Rocq seria com uma propriedade entre duas listas de inteiros, que faz a tradução mais ingênua possível:

```

Inductive naive_utf8_map : byte_str -> codepoint -> Prop :=
| OneByte (b1: byte) :
  0x00 <= b1 < 0x80 ->
  naive_utf8_map [b1] b1
| TwoBytes (b1 b2: byte) :
  0xc2 <= b1 <= 0xdf ->
  0x80 <= b2 <= 0xbf ->
  naive_utf8_map [b1; b2] ((b1 mod 64) * 64 + (b2 mod 64))
| ThreeBytes1 (b1 b2 b3: Z):
  b1 = 0xe0 ->
  0xa0 <= b2 <= 0xbf ->
  0x80 <= b3 <= 0xbf ->
  naive_utf8_map [b1; b2; b3] (((b1 - 224) * 4096) + (b2 mod 64) * 64 + (b3 mod 64))
| ThreeBytes2 (b1 b2 b3: Z):
  0xe1 <= b1 <= 0xec \ / 0xee <= b1 <= 0xef ->
  0x80 <= b2 <= 0xbf ->
  0x80 <= b3 <= 0xbf ->
  naive_utf8_map [b1; b2; b3] (((b1 - 224) * 4096) + (b2 mod 64) * 64 + (b3 mod 64))
| ThreeBytes3 (b1 b2 b3: Z):
  b1 = 0xed ->
  0x80 <= b2 <= 0x9f ->
  0x80 <= b3 <= 0xbf ->
  naive_utf8_map [b1; b2; b3] (((b1 - 224) * 4096) + (b2 mod 64) * 64 + (b3 mod 64))

```

```
| FourBytes1 (b1 b2 b3 b4: Z):
  b1 = 0xf0 ->
  0x90 <= b2 <= 0xbf ->
  0x80 <= b3 <= 0xbf ->
  0x80 <= b4 <= 0xbf ->
  naive_utf8_map [b1; b2; b3; b4] ((b1 - 240) * 262144 + (b2 mod 64) * 4096 + (b3 mod
64) * 64 + (b4 mod 64))
| FourBytes2 (b1 b2 b3 b4: Z):
  0xf1 <= b1 <= 0xf3 ->
  0x80 <= b2 <= 0xbf ->
  0x80 <= b3 <= 0xbf ->
  0x80 <= b4 <= 0xbf ->
  naive_utf8_map [b1; b2; b3; b4] ((b1 - 240) * 262144 + (b2 mod 64) * 4096 + (b3 mod
64) * 64 + (b4 mod 64))
| FourBytes3 (b1 b2 b3 b4: Z):
  b1 = 0xf4 ->
  0x80 <= b2 <= 0x8f ->
  0x80 <= b3 <= 0xbf ->
  0x80 <= b4 <= 0xbf ->
  naive_utf8_map [b1; b2; b3; b4] ((b1 - 240) * 262144 + (b2 mod 64) * 4096 + (b3 mod
64) * 64 + (b4 mod 64)).
```

Isto é, um elemento de tipo `naive_utf8_map bytes codepoint` é uma prova de que a sequência de bytes `bytes` mapeia para o codepoint `codepoint` segundo as tabelas Tabela 3 e Tabela 5. Especificamente, cada construtor de `naive_utf8_map` representa uma das linhas da Tabela 5, e as operações nos bytes de multiplicação e `mod` representam como extrair os bits relevantes dos bytes que contém cabeçalhos.

Entretanto, o problema dessa especificação é que não há como afirmar com certeza que essas operações representam exatamente o que é dado na Tabela 3, visto que há muitas operações envolvidas. Parte crucial de verificação de software é que a especificação seja simples de entender, para que essa seja trivialmente verificável, e infelizmente essa tabela não é facilmente compreendida.

Assim, esse tipo não será utilizado. Ao invés de especificar exatamente qual o mapeamento dado entre bytes e codepoints, é mais interessante considerar propriedades que esse deve satisfazer. Especificamente, é simples explicitar as propriedades que ditam o que é uma sequência de bytes UTF-8 válidas (Tabela 5) e o que é um *code point* válido, exigindo que o codificador mapeie *code points* válidos em bytes UTF-8 válidos, e o decodificador mapeie bytes UTF-8 válidos em *code points* válidos. Entretanto, existem inúmeras maneiras de fazer esse mapeamento de modo que o codificador e decodificador sejam inversos, e apenas um desses de fato é o UTF-8.

Para especificar **como** *code points* são mapeados em bytes, a seguinte propriedade denotada no RFC 3629 é extremamente útil:

“A ordenação lexicográfica por valor dos bytes de strings UTF-8 é a mesma que se fosse ordenada pelos números dos caracteres. É claro, isso é de interesse limitado, dado que uma ordenação baseada no número dos caracteres quase nunca é culturalmente válida.” (YER-GEAU (2003))

Apesar do que foi dito pelo autor do RFC, essa propriedade é de extremo interesse para a formalização por sua simplicidade. Para garantir que *code points* sejam mapeados nas respectivas

representações de bytes, basta exigir que tanto o codificador quanto o decodificador respeitem a ordenação lexicográfica entre *code points* e bytes.

Assim, são definidos as seguintes notações:

**Definition** `codepoint` : `Type` := `Z`.

**Definition** `byte` : `Type` := `Z`.

**Definition** `unicode_str` : `Type` := `list codepoint`.

**Definition** `byte_str` : `Type` := `list byte`.

**Definition** `codepoints_compare` := `List.list_compare Z.compare`.

**Definition** `bytes_compare` := `List.list_compare Z.compare`.

As funções `codepoints_compare` e `bytes_compare` são utilizadas exatamente para prover as comparações entre inteiros. A função `Z.compare` é oferecida pela biblioteca padrão do Rocq, recebendo dois inteiros e retorna o resultado da comparação entre eles, do tipo `comparison`:

**Inductive** `comparison` : `Set` :=

| `Eq` : `comparison`

| `Lt` : `comparison`

| `Gt` : `comparison`.

A função `list_compare` transforma uma comparação entre elementos de um tipo `T` em uma comparação entre elementos de tipo `list T`, utilizando a semântica de comparação lexicográfica.

Em seguida, são definidas as propriedades necessárias para afirmar que um `codepoint` arbitrário, isto é, um inteiro qualquer, é um *codepoint* UTF-8 válido. Como visto anteriormente, basta saber que esse está entre 0 e 10FFFF, e não está no intervalo D800..DFFF. Isso pode ser representado como as seguintes três propriedades:

**Definition** `codepoint_less_than_10ffff` (`code`: `codepoint`) : `Prop` :=  
(`code` <= 0x10ffff).

**Definition** `codepoint_is_not_surrogate` (`code`: `codepoint`) : `Prop` :=  
(`code` < 0xd800) ∨ (`code` > 0xdfff).

**Definition** `codepoint_not_negative` (`code`: `codepoint`): `Prop` :=  
(`code` >= 0).

**Definition** `valid_codepoint` (`code`: `codepoint`) := `codepoint_less_than_10ffff code` ∧  
`codepoint_is_not_surrogate code` ∧ `codepoint_not_negative code`.

Isto é, provar que `valid_codepoint code` para algum `code` significa mostrar que as três propriedades valem ao mesmo tempo.

Para definir o tipo `valid_codepoint_representation`, será utilizada a mesma ideia do `naive_utf8_map`. Isto é, esse só pode ser construído quando os elementos da lista de entrada estão nos intervalos de alguma das linhas da tabela, e representa afirmar que uma certa lista de bytes é a representação em UTF-8 de algum *codepoint*:

**Inductive** `valid_codepoint_representation` : `list Z -> Prop` :=

| `OneByte` (`b`: `Z`) :

0 <= `b` <= 0x7f ->

`valid_codepoint_representation` [`b`]

| `TwoByte` (`b1 b2`: `Z`):

0xc2 <= `b1` <= 0xdf ->

0x80 <= `b2` <= 0xbf ->

`valid_codepoint_representation` [`b1`; `b2`]

```

| ThreeByte1 (b1 b2 b3: Z):
  b1 = 0xe0 ->
  0xa0 <= b2 <= 0xbf ->
  0x80 <= b3 <= 0xbf ->
  valid_codepoint_representation [b1; b2; b3]
| ThreeByte2 (b1 b2 b3: Z):
  0xe1 <= b1 <= 0xec \ / 0xee <= b1 <= 0xef ->
  0x80 <= b2 <= 0xbf ->
  0x80 <= b3 <= 0xbf ->
  valid_codepoint_representation [b1; b2; b3]
| ThreeByte3 (b1 b2 b3: Z):
  b1 = 0xed ->
  0x80 <= b2 <= 0x9f ->
  0x80 <= b3 <= 0xbf ->
  valid_codepoint_representation [b1; b2; b3]
| FourBytes1 (b1 b2 b3 b4: Z):
  b1 = 0xf0 ->
  0x90 <= b2 <= 0xbf ->
  0x80 <= b3 <= 0xbf ->
  0x80 <= b4 <= 0xbf ->
  valid_codepoint_representation [b1; b2; b3; b4]
| FourBytes2 (b1 b2 b3 b4: Z):
  0xf1 <= b1 <= 0xf3 ->
  0x80 <= b2 <= 0xbf ->
  0x80 <= b3 <= 0xbf ->
  0x80 <= b4 <= 0xbf ->
  valid_codepoint_representation [b1; b2; b3; b4]
| FourBytes3 (b1 b2 b3 b4: Z):
  b1 = 0xf4 ->
  0x80 <= b2 <= 0x8f ->
  0x80 <= b3 <= 0xbf ->
  0x80 <= b4 <= 0xbf ->
  valid_codepoint_representation [b1; b2; b3; b4].
    
```

Com isso, existem duas maneiras de construir uma lista de bytes válidos UTF-8: ou a lista é vazia, ou ela é a concatenação de uma representação em bytes de um codepoint e uma lista de bytes UTF-8 válidos. O tipo que representa que essa relação é:

```

Inductive valid_utf8_bytes: list Z -> Prop :=
| Utf8Nil : valid_utf8_bytes []
| Utf8Concat (bytes tail: list Z) :
  valid_codepoint_representation bytes ->
  valid_utf8_bytes tail ->
  valid_utf8_bytes (bytes ++ tail).
    
```

Apenas essas definições são suficientes para começar a definir as propriedades que o codificador e decodificador devem seguir:

**Definition** `encoder_nil` (encoder: `encoder_type`) := encoder [] = ([], []).

A primeira propriedade dita que o `encoder` deve aceitar a lista vazia com o resultado vazio.

**Definition** `encoder_input_correct_iff` (encoder: `encoder_type`) := forall code,  
 valid\_codepoint code <->  
 exists bytes, encoder [code] = (bytes, []).



A segunda propriedade é uma dupla implicação: da esquerda para direita, diz que o `encoder` deve aceitar todo `codepoint` válido; da direita para esquerda, diz que se o `encoder` aceita uma lista com um `codepoint` apenas, então esse `codepoint` é válido.

```
Definition encoder_output_correct (encoder: encoder_type) := forall code,
  match encoder [code] with
  | (bytes, []) => valid_codepoint_representation bytes
  | (bytes, rest) => bytes = [] /\ rest = [code]
  end.
```

A terceira propriedade descreve sobre a validade do resultado de um `encoder`. Apenas dois resultados ao chamar um `encoder` com uma lista de um elemento são possíveis: ou a entrada é aceita, e os `bytes` à esquerda são uma representação de `codepoints` válida, ou não é aceita, o que implica que os `bytes` devem ser vazios, e o lado não consumido deve conter o `codepoint` da entrada.

```
Definition encoder_strictly_increasing (encoder: encoder_type) := forall codes1
codes2 bytes1 bytes2,
  encoder codes1 = (bytes1, nil) ->
  encoder codes2 = (bytes2, nil) ->
  codepoints_compare codes1 codes2 = bytes_compare bytes1 bytes2.
```

A quarta propriedade afirma que o `encoder` respeita a ordenação lexicográfica entre `bytes` e `code points`, explicada anteriormente. Essa propriedade é suficiente para afirmar que o `encoder` mapeia o `code point` na sua respectiva representação em `bytes`, de acordo com o mapeamento UTF-8.

```
Definition encoder_projects (encoder: encoder_type) := forall xs ys,
  encoder (xs ++ ys) =
  match encoder xs with
  | (bytes, nil) =>
    let (bytes2, rest) := encoder ys in
    (bytes ++ bytes2, rest)
  | (bytes, rest) => (bytes, rest ++ ys)
  end.
```

Por fim, a quinta e última propriedade é a que descreve como o `encoder` deve se comportar perante listas grandes. Quando uma lista pode ser quebrada em duas listas menores, o resultado de chamar o `encoder` na lista maior é igual a chamar na primeira, e se for aceita, chamar na segunda e concatenar os resultados. No caso de erro, o `encoder` para imediatamente.

```
Record utf8_encoder_spec encoder := {
  enc_nil : encoder_nil encoder;
  enc_increasing : encoder_strictly_increasing encoder;
  enc_input : encoder_input_correct_iff encoder;
  enc_output : encoder_output_correct encoder;
  enc_projects : encoder_projects encoder;
}.
```

Apenas essas 5 propriedades são o suficiente para qualificar um `encoder` como um codificador de UTF-8 válido, segundo a especificação. Importantemente, não é necessário ter um decodificador para provar que o codificador está correto. Para provar que um `encoder` está certo, basta construir um elemento de tipo `utf8_encoder_spec encoder`.

As propriedades que o decodificador deve satisfazer são análogas às do codificador.

```
Definition decoder_nil (decoder: decoder_type) := decoder nil = (nil, nil).
```

**Definition** `decoder_output_correct` (decoder: `decoder_type`) := forall bytes suffix codes,

```

    decoder bytes = (codes, suffix) ->
    valid_codepoints codes /\
    (exists prefix,
        decoder prefix = (codes, [])
        /\ valid_utf8_bytes prefix
        /\ bytes = prefix ++ suffix).
```

**Definition** `decoder_input_correct_iff` (decoder: `decoder_type`) := forall bytes, valid\_codepoint\_representation bytes <-> exists code, decoder bytes = ([code], []).

**Definition** `decoder_strictly_increasing` (decoder: `decoder_type`) := forall bytes1 bytes2 code1 code2, decoder bytes1 = ([code1], nil) -> decoder bytes2 = ([code2], nil) -> Z.compare code1 code2 = bytes\_compare bytes1 bytes2.

**Definition** `decoder_projects` (decoder: `decoder_type`) := forall xs ys, valid\_codepoint\_representation xs -> decoder (xs ++ ys) = let (codes, \_) := decoder xs in let (codes2, rest) := decoder ys in (codes ++ codes2, rest).

```

Record utf8_decoder_spec decoder := {
    dec_nil : decoder_nil decoder;
    dec_input : decoder_input_correct_iff decoder;
    dec_output : decoder_output_correct decoder;
    dec_increasing : decoder_strictly_increasing decoder;
    dec_projects : decoder_projects decoder;
}.
```

A primeira propriedade afirma que todo `decoder` aceita a lista vazia. A segunda afirma que do *code point* emitido pelo `decoder` deve ser válido. A terceira fala que todo input válido deve ser aceito. A quarta propriedade afirma sobre a ordenação entre bytes e *code points*, assim como no `decoder`. A quinta propriedade é uma propriedade de projeção para desconstruir listas em listas menores.

Com essas duas definições, a especificação UTF-8 completa para um par codificador e decodificador é o par que contém a especificação para o codificador e decodificador separadamente. Por serem separados, é possível mostrar que quaisquer `encoder` e `decoder` são corretos, contanto que mostre que as regras valem para eles separadamente.

```

Record utf8_spec encoder decoder := {
    encoder_spec_compliant : utf8_encoder_spec encoder;
    decoder_spec_compliant : utf8_decoder_spec decoder;
}.
```

## 4.1 Corretude da especificação

Para ter certeza de que a especificação está correta, é necessário provar teoremas sobre ela. Como visto anteriormente, a propriedades principal que formará o cerne da corretude da especificação é de que todo par (`encoder`, `decoder`) que implemente `utf8_spec encoder decoder` deve necessariamente ser inverso um do outro. Por ambos o codificador e decodificador serem

um *parser* parcial, é preciso considerar que nem toda entrada irá ser aceita, e isso é levado em conta da seguinte forma: toda entrada deve necessariamente ter um prefixo UTF-8 válido – que pode ser a lista vazia – de forma que o prefixo válido deve ser a entrada para o processador dual.

**Theorem** `utf8_spec_encoder_decoder_inverse` : `forall` encoder decoder,  
`utf8_spec encoder decoder ->`  
`forall` codes bytes codes\_suffix,  
`encoder codes = (bytes, codes_suffix) ->`  
`exists` codes\_prefix, decoder bytes = (codes\_prefix, nil) /\ codes =  
(codes\_prefix ++ codes\_suffix)%list.

**Theorem** `utf8_spec_decoder_encoder_inverse_strong` : `forall` encoder decoder,  
`utf8_spec encoder decoder ->`  
`forall` codes bytes bytes\_suffix,  
`decoder bytes = (codes, bytes_suffix) ->`  
`exists` bytes\_prefix, encoder codes = (bytes\_prefix, nil) /\ bytes =  
(bytes\_prefix ++ bytes\_suffix)%list.

**Proof.**

Isto é, se `encoder codes = (bytes, codes_suffix)`, então necessariamente deve existir um prefixo `codes_prefix` tal que `decoder bytes = (codes_prefix, [])` e `codes = codes_prefix + codes_suffix`.

Para provar essas propriedades, muito trabalho é necessário. Intuitivamente, a prova é inteiramente baseada no fato de que ordenação implica em existir apenas uma função que respeite o mapeamento entre bytes e *code points*, entretanto isso não é nem um pouco óbvio. Assim, é necessário mostrar esse fato para que possa ser utilizado nas provas seguintes.

## 4.2 Ordenações em conjuntos finitos

Tanto `valid_codepoint` quanto `valid_codepoint_representation` são propriedades que formam conjuntos finitos de exato mesmo tamanho (`10FFFF - 0x800` elementos, o número de *code points*). Por serem conjuntos finitos, é possível assinalar um inteiro para cada elemento. Assim, provar que são equivalentes significa provar que a necessidade de respeitar ordenação implica que existe apenas um mapeamento entre conjuntos finitos de mesmo tamanho.

Isto é, é possível mapear cada *code point* e cada sequência de *bytes* em um único inteiro unicamente, utilizando a ordenação natural dos inteiros, construindo funções de `nth_valid_codepoint` – que retorna o *n*ésimo codepoint – e `nth_valid_codepoint_representation` – que retorna a sequência de bytes do *n*ésimo codepoint. Além disso, ao provar que essas funções tem inversa (isto é, fornecer uma função que recebe um inteiro e retorna o *code point*/sequência de bytes equivalente), fica claro que ambas essas funções formam isomorfismos nesse conjunto de inteiros, ambas respeitando a ordenação.

É um fato da matemática todo isomorfismo entre os mesmos dois conjuntos totalmente ordenados é único, e portanto deveria ser possível mostrar que a composição de dois desses isomorfismos é única. Desse fato é derivável que há um isomorfismo único entre `valid_codepoint` e `valid_codepoint_representation`, na ida compondo `inverse_nth_valid_codepoint` com `nth_valid_codepoint_representation`, e na volta compondo `nth_valid_codepoint` com `inverse_nth_valid_codepoint_representation`. Entretanto, a composição de codificador e decodificador também formam isomorfismos entre os mesmos conjuntos, e pela unicidade devem necessariamente serem iguais à fazer a tradução utilizando os índices.

Para formalizar essa noção, são definidos morfismos parciais:

```
Definition interval (count n : Z) : Prop :=
  (0 <= n /\ n < count)%Z.
```

```
Definition partial_morphism {X Y}
  (domain : X -> Prop) (range : Y -> Prop) (f : X -> option Y) : Prop :=
  (forall (x : X) (y : Y), f x = Some y -> range y) (* f is contained in the range *)
  /\ (forall (x : X), f x = None -> (not (domain x))) (* f always returns a value in
  its domain *).
```

```
Definition and_then {X Y Z}
  (f : X -> option Y) (g : Y -> option Z) : X -> option Z :=
  fun x =>
    match (f x) with
    | Some y => (g y)
    | None => None
  end.
```

```
Definition pointwise_equal {X Y}
  (domain : X -> Prop) (f g : X -> option Y) : Prop :=
  forall x, domain x -> f x = g x.
```

Como motivação, é fácil ver que o codificador com `valid_codepoint` forma um morfismo parcial (de  $Z$  em `valid_codepoint`), bem como o decodificador com `valid_codepoint_representation`. A definição `pointwise_equal f g` é utilizada no lugar da igualdade  $f = g$ , pois provar igualdade de funções em Coq a partir da igualdade de elementos não é possível; isto é, não é possível provar que  $f = g$  com a hipótese de que `pointwise_equal f g` sem adicionar axiomas externos (extensionalidade funcional).

Além disso, é definida a noção de conjunto ordenado:

```
Record Ordered {T} (compare: T -> T -> comparison) := {
  eq : forall t1 t2, compare t1 t2 = Eq <-> t1 = t2;
  antisym : forall t1 t2, compare t1 t2 = CompOpp (compare t2 t1);
  trans : forall t1 t2 t3 res, compare t1 t2 = res -> compare t2 t3 = res ->
  compare t1 t3 = res;
}.
```

Para prova provar que um tipo  $T$  é ordenado, basta mostrar que existe uma relação de comparação reflexiva, antisimétrica e transitiva. Além disso, é caracterizada a noção de ser “crescente” da seguinte forma:

```
Definition increasing {T1 T2}
  (domain: T1 -> Prop)
  (compare1: T1 -> T1 -> comparison) (compare2: T2 -> T2 -> comparison)
  (to: T1 -> option T2) :=
  forall n m, (domain n) -> (domain m) ->
    match (to n, to m) with
    | (Some a, Some b) => (compare1 n m) = (compare2 a b)
    | _ => False
  end.
```

Informalmente, uma função  $f$  é `increasing` se `compare1 a b = compare2 (f a) (f b)`, ou seja, se respeita a comparação entre quaisquer dois elementos. Com isso, finalmente pode-se definir o que é um isomorfismo ordenado:

```
Record OrderedPartialIsomorphism {T1 T2} (domain: T1 -> Prop) (range: T2 -> Prop)
  (compare1: T1 -> T1 -> comparison) (compare2: T2 -> T2 -> comparison) (to: T1 ->
```

```
option T2) (from: T2 -> option T1)
:= {
  ordered1 : @Ordered T1 compare1;
  ordered2 : @Ordered T2 compare2;
  from_morphism : partial_morphism domain range to;
  to_morphism : partial_morphism range domain from;
  from_to_id : pointwise_equal domain (and_then to from) (fun x => Some x);
  to_from_id : pointwise_equal range (and_then from to) (fun x => Some x);
  from_preserves_compare : increasing domain compare1 compare2 to;
}.
```

Um isomorfismo ordenado é um par de funções `from` e `to` que mapeiam entre conjuntos ordenados `T1` e `T2`, de forma que a composição deles dá a identidade. Além disso, é necessário mostrar que ambos formam morfismos entre seu respectivo domínio e imagem, e que pelo menos um deles é `increasing` – por simplicidade, o `from`.

Assim, o teorema principal de ordenação pode ser enunciado:

```
Theorem partial_isomorphism_countable_unique {T0 T1} (count: Z) (range0: T0 -> Prop)
(range1: T1 -> Prop) compare0 compare1:
  forall from0 from1 from2 to0 to1 to2,
    OrderedPartialIsomorphism (interval count) range0 Z.compare compare0 to0 from0 ->
    OrderedPartialIsomorphism (interval count) range1 Z.compare compare1 to1 from1 ->
    partial_morphism range0 range1 to2 ->
    partial_morphism range1 range0 from2 ->
    increasing range0 compare0 compare1 to2 ->
    increasing range1 compare1 compare0 from2 ->
    (pointwise_equal range0 to2 (and_then from0 to1))
  /\ (pointwise_equal range1 from2 (and_then from1 to0)).
```

Esse teorema permite afirmar que compor qualquer morfismo parcial entre `valid_codepoint` e `valid_codepoint_representation` que respeite a ordenação deve necessariamente ser igual (no sentido de `pointwise_equal`) a compor as operações de índice (`nth_valid_codepoint` e `nth_valid_codepoint_representation`). Com isso, torna-se possível derivar que todo `encoder` e `decoder` que respeita ordenação deve concordar em todos os valores.

Assim, para usar esse teorema é necessário definir `nth_valid_codepoint_representation` e `nth_valid_codepoint` e mostrar que ambos formam isomorfismos parciais com o conjunto de inteiros entre 0 e `0x10FFFF - 0x800`.

### 4.3 Índices de codepoints e de sequências de bytes

Para definir as funções supracitadas, é necessário lembrar que o conjunto de índices exclui codepoints no intervalo `0xD800..0xDFFF`, ou seja, o índice deve “pular” esse intervalo. Assim, a única preocupação da função `nth_valid_codepoint` é somar `0x800` quando isso acontece:

```
Definition nth_valid_codepoint (n: Z) : option codepoint :=
  if n <? 0 then
    None
  else if n <? 0xD800 then
    Some n
  else if n <=? 0x10ffff - 0x0800 then
    Some (n + 0x0800)
  else
    None.
```

Para mostrar que essa função forma um isomorfismo parcial, as seguintes propriedades são provadas:

```
Lemma nth_valid_codepoint_is_some_iff_valid : forall code,
  (exists n, nth_valid_codepoint n = Some code) <->
  valid_codepoint code.
```

```
Lemma nth_valid_codepoint_none : forall n,
  nth_valid_codepoint n = None ->
  n < 0 \\/ n > (0x10ffff - 0x800).
```

```
Lemma nth_valid_codepoint_increasing : forall n1 code1 n2 code2,
  nth_valid_codepoint n1 = Some code1 ->
  nth_valid_codepoint n2 = Some code2 ->
  Z.compare n1 n2 = Z.compare code1 code2.
```

A prova desses teoremas é omitida por brevidade, mas todas envolvem observar as comparações feitas em `nth_valid_codepoint` e utilizar a tática `lia` para casos específicos, que resolve relações na aritmética de Presburgo. Em especial, a prova de que respeita a comparação é feita considerando todas as possíveis maneiras que os ifs podem se desdobrar, e mostrar que em todas elas as comparações são iguais.

Além disso, é necessário oferecer a função inversa dessa, que vai do índice do codepoint para o codepoint:

```
Definition inverse_nth_valid_codepoint (code: codepoint) : option Z :=
  if (code <? 0) then
    None
  else if (code <? 0xd800) then
    Some code
  else if (code <=? 0x10ffff)%Z then
    Some (code - 0x0800)%Z
  else
    None.
```

Bem como provar que ambas são inversas:

```
Lemma nth_valid_codepoint_invertible : forall code n,
  nth_valid_codepoint n = Some code <->
  inverse_nth_valid_codepoint code = Some n /\ valid_codepoint code.
```

Assim, é possível provar que essa função forma um isomorfismo parcial ordenado, construindo um elemento do seguinte tipo:

```
Definition codepoint_nth_isomorphism : OrderedPartialIsomorphism (interval (0x10ffff
- 0x7ff)) valid_codepoint Z.compare codepoint_compare nth_valid_codepoint
inverse_nth_valid_codepoint.
```

Recapitulando, `codepoint_nth_isomorphism` é a prova de que o par  $(\text{nth\_valid\_codepoint}, \text{inverse\_nth\_valid\_codepoint})$  formam um isomorfismo com o conjunto de índices, e esse isomorfismo respeita a ordenação de codepoints e a ordenação de índices. A construção dessa prova utiliza todos os lemmas supracitados, bem como a prova de que o conjunto dos inteiros é um conjunto ordenado:

```
Definition ZOrder : @Ordered Z Z.compare.
  split. apply Z.compare_eq_iff. intros. apply Z.compare_antisym.
  intros. destruct res.
  - apply Z.compare_eq_iff in H, H0. subst. apply Z.compare_refl.
```

- apply Zcompare.Zcompare\_Lt\_trans with (m := t2); assumption.  
 - apply Zcompare.Zcompare\_Gt\_trans with (m := t2); assumption.  
 Qed.

Após isso, é necessário definir o mesmo para nth\_valid\_code\_representation.

```
Definition nth_valid_codepoint_representation (n: Z) : option byte_str :=
  let n := if Z.ltb n 0xd800 then n else n + 0x800 in
  if (n <? 0) then
    None
  else if (n <=? 127) then
    Some [ n ]
  else if (n <=? 0x7fff) then
    let b1 := n / 64 in
    let b2 := n mod 64 in
    Some [ 192 + b1; 128 + b2 ]
  else if (n <=? 0xffff) then
    let r := n / 64 in
    let b1 := r / 64 in
    let b2 := r mod 64 in
    let b3 := n mod 64 in
    Some [ 224 + b1; 128 + b2; 128 + b3 ]
  else if (n <=? 0x10ffff) then
    let r1 := n / 64 in
    let r2 := r1 / 64 in
    let b1 := r2 / 64 in
    let b2 := r2 mod 64 in
    let b3 := r1 mod 64 in
    let b4 := n mod 64 in
    Some [ 240 + b1; 128 + b2; 128 + b3; 128 + b4 ]
  else
    None.
```

E provar os mesmos lemmas:

```
Lemma nth_valid_codepoint_representation_spec: forall bytes,
  (exists n, nth_valid_codepoint_representation n = Some bytes) <->
  valid_codepoint_representation bytes.
```

```
Lemma nth_valid_codepoint_representation_none : forall n : Z,
  nth_valid_codepoint_representation n = None ->
  n < 0 \\/ n > (1114111 - 2048).
```

```
Lemma nth_valid_codepoint_representation_compare_compat: forall n1 n2 bytes1 bytes2,
  nth_valid_codepoint_representation n1 = Some bytes1 ->
  nth_valid_codepoint_representation n2 = Some bytes2 ->
  Z.compare n1 n2 = bytes_compare bytes1 bytes2.
```

A prova desses é significativamente mais complexa, pois a função que mapeia o índice na sequência de bytes equivalente é muito mais complexa. Para facilitar a análise, táticas especiais foram criadas para automatizar a resolução de casos parecidos utilizando a tática *lia*.

Também é necessário desenvolver a função que calcula o índice do codepoint a partir da sequência de bytes.

```
Definition inverse_nth_valid_codepoint_representation (bytes: byte_str) : option Z :=
  let between b lo hi := andb (lo <=? b) (b <=? hi) in
  match bytes with
```

```

| [b] => if between b 0 127 then Some b else None
| [b1; b2] =>
  if andb (between b1 0xc2 0xdf) (between b2 0x80 0xbf) then
    Some ((b1 mod 64) * 64 + (b2 mod 64))
  else None
| [b1; b2; b3] =>
  let fst := andb (andb (b1 =? 0xe0) (between b2 0xa0 0xbf)) (between b3 0x80
0xbf) in
  let snd := andb (andb (between b1 0xe1 0xec) (between b2 0x80 0xbf)) (between
b3 0x80 0xbf) in
  let trd := andb (andb (b1 =? 0xed) (between b2 0x80 0x9f)) (between b3 0x80
0xbf) in
  let frth := andb (andb (between b1 0xee 0xef) (between b2 0x80 0xbf)) (between
b3 0x80 0xbf) in
  let n := ((b1 - 224) * 64 * 64) + (b2 mod 64) * 64 + (b3 mod 64) in
  if orb (orb fst snd) trd then
    Some n
  else if frth then
    Some (n - 2048)
  else
    None
| [b1; b2; b3; b4] =>
  let fst := andb (andb (andb (b1 =? 0xf0) (between b2 0x90 0xbf)) (between b3
0x80 0xbf)) (between b4 0x80 0xbf) in
  let snd := andb (andb (andb (between b1 0xf1 0xf3) (between b2 0x80 0xbf))
(between b3 0x80 0xbf)) (between b4 0x80 0xbf) in
  let trd := andb (andb (andb (b1 =? 0xf4) (between b2 0x80 0x8f)) (between b3
0x80 0xbf)) (between b4 0x80 0xbf) in
  if orb (orb fst snd) trd then
    Some ((b1 - 240) * 64 * 64 * 64 + (b2 mod 64) * 64 * 64 + (b3 mod 64) * 64 +
(b4 mod 64) - 0x800)
  else None
| _ => None
end.

```

Vale notar que as operações que essa executa são exatamente as mesmas operações dadas em `naive_utf8_map`, mas dessa vez, a corretude dessas operações é checada no fato de que essa é a inversa da `nth_valid_codepoint_representation`:

**Lemma** `nth_valid_codepoint_representation_invertible` : `forall` n bytes,  
`nth_valid_codepoint_representation n = Some bytes ->`  
`inverse_nth_valid_codepoint_representation bytes = Some n.`

**Lemma** `inverse_nth_valid_codepoint_representation_invertible` : `forall` bytes n,  
`valid_codepoint_representation bytes ->`  
`inverse_nth_valid_codepoint_representation bytes = Some n ->`  
`nth_valid_codepoint_representation n = Some bytes.`

Por fim, também é necessário provar que o conjunto de sequências de bytes é ordenado, de acordo com a comparação lexicográfica.

**Definition** `BytesOrder` : `Ordered bytes_compare.`

**Proof.**

```

  unfold bytes_compare.
  split.
  - apply list_compare_refl. apply Z.compare_eq_iff.
  - intros.

```



```

    apply list_compare_antisym. apply Z.compare_eq_iff. apply Z.compare_antisym.
- intros.
    apply list_compare_trans with (ys:=t2); try assumption.
+ apply Z.compare_eq_iff.
+ intros. destruct c.
  -- apply Z.compare_eq_iff in H1, H2. subst. apply Z.compare_refl.
  -- apply Zcompare.Zcompare_Lt_trans with (m := y); assumption.
  -- apply Zcompare.Zcompare_Gt_trans with (m := y); assumption.
+ apply Z.compare_antisym.
Qed.

```

Assim, é possível provar que o par  $(\text{nth\_valid\_codepoint\_representation}, \text{inverse\_nth\_valid\_codepoint\_representation})$  forma um isomorfismo com o conjunto dos inteiros de  $0x10ffff - 0x7ff$ , e que esse isomorfismo respeita a ordenação:

```

Theorem valid_codepoint_representation_isomorphism :
  OrderedPartialIsomorphism (interval (0x10ffff - 0x7ff))
  valid_codepoint_representation Z.compare bytes_compare
  nth_valid_codepoint_representation inverse_nth_valid_codepoint_representation.

```

#### 4.4 Especificação implica mapeamento UTF-8 correto

O objetivo de mostrar essas propriedades de ordenação e de índice é utilizar `partial_isomorphism_countable_unique` para provar os seguintes teoremas:

```

Theorem utf8_spec_implies_encoder_maps_nth_to_nth : forall encoder decoder,
  utf8_spec encoder decoder ->
  forall code bytes,
    encoder [code] = (bytes, []) ->
    exists n, nth_valid_codepoint n = Some code /\
  nth_valid_codepoint_representation n = Some bytes.

```

```

Lemma utf8_spec_implies_decoder_maps_nth_to_nth : forall encoder decoder,
  utf8_spec encoder decoder ->
  forall code bytes,
    decoder bytes = ([code], []) ->
    exists n, nth_valid_codepoint n = Some code /\
  nth_valid_codepoint_representation n = Some bytes.

```

Isto é, quando um codificador aceita um codepoint, então o resultado é a sequência de bytes com o índice equivalente. Da mesma forma, quando o decodificador aceita uma sequência de bytes, então o resultado é o codepoint com o índice equivalente.

Para utilizar `partial_isomorphism_countable_unique` nessa prova, é necessário construir morfismos parciais (que retornam `option` ao invés de listas) a partir de codificadores e decodificadores:

```

Definition encoder_to_option (encoder: encoder_type) code :=
  match encoder [code] with
  | (bytes, []) => Some bytes
  | _ => None
  end.

```

```

Definition decoder_to_option (decoder: decoder_type) bytes :=
  match decoder bytes with
  | ([code], []) => Some code
  | _ => None
  end.

```

Assim, os seguintes lemmas sobre encoder e decoder são provados, para que possam ser utilizados nas provas:

```
Lemma encoder_partial_morphism : forall encoder,
  utf8_encoder_spec encoder ->
  partial_morphism valid_codepoint valid_codepoint_representation
  (encoder_to_option encoder).
```

```
Lemma decoder_partial_morphism : forall decoder,
  utf8_decoder_spec decoder ->
  partial_morphism valid_codepoint_representation valid_codepoint
  (decoder_to_option decoder).
```

```
Lemma encoder_to_option_increasing : forall encoder,
  utf8_encoder_spec encoder ->
  increasing valid_codepoint Z.compare bytes_compare (encoder_to_option encoder).
```

```
Lemma decoder_to_option_increasing : forall decoder,
  utf8_decoder_spec decoder ->
  increasing valid_codepoint_representation bytes_compare Z.compare
  (decoder_to_option decoder).
```

Com os lemmas de mapeamento de  $n$  pra  $n$  em mãos, é trivial mostrar que tanto o encoder quanto o decoder devem ser inversos no caso de apenas um codepoint:

```
Theorem utf8_spec_decoder_encoder_inverse_single : forall encoder decoder,
  utf8_encoder_spec encoder ->
  utf8_decoder_spec decoder ->
  forall code bytes,
    decoder bytes = ([code], []) ->
    encoder [code] = (bytes, []).
```

**Proof.**

```
intros encoder decoder encoder_spec decoder_spec.
intros code bytes decode_bytes.
eapply utf8_spec_implies_decoder_maps_nth_to_nth in decode_bytes as G; [ | apply
encoder_spec | assumption].
destruct G as [n [nth_code nth_byte]].
apply dec_output in decode_bytes as [valid_code _]; [|assumption].
eapply encoder_encode_valid_codepoints in valid_code; [| apply encoder_spec].
destruct valid_code as [bytes2 [encoder_code _]].
eapply utf8_spec_implies_encoder_maps_nth_to_nth in encoder_code as G; [ | apply
encoder_spec | apply decoder_spec].
destruct G as [n2 [nth2_code nth2_byte]].
apply nth_valid_codepoint_invertible in nth_code as [inverse_n _], nth2_code as
[inverse_n2 _].
rewrite inverse_n in inverse_n2. apply some_injective in inverse_n2.
subst. rewrite nth2_byte in nth_byte. apply some_injective in nth_byte.
subst. assumption.
```

**Qed.**

```
Theorem utf8_spec_encoder_decoder_inverse : forall encoder decoder,
  utf8_encoder_spec encoder ->
  utf8_decoder_spec decoder ->
  forall codes bytes codes_suffix,
    encoder codes = (bytes, codes_suffix) ->
    exists codes_prefix, decoder bytes = (codes_prefix, nil) /\ codes =
(codes_prefix ++ codes_suffix)%list.
```

**Proof.**

```

intros encoder decoder encoder_spec decoder_spec.
induction codes as [| code tail]; intros bytes codes_suffix encode_codes.
- exists []. pose proof (enc_nil encoder encoder_spec). rewrite H in encode_codes.
inversion encode_codes.
split. apply dec_nil. assumption. reflexivity.
- replace (code :: tail) with ([code] ++ tail) in encode_codes by reflexivity.
rewrite enc_projects in encode_codes; [| assumption].
destruct (encoder [code]) as [bytes2 rest] eqn:encoder_code.
destruct rest.
2: {
  inversion encode_codes. subst.
  specialize (enc_output encoder encoder_spec code) as bytes_invalid.
  rewrite encoder_code in bytes_invalid. destruct bytes_invalid as [bytes_eq
rest_eq].
  inversion rest_eq. subst.
  exists nil. split. apply dec_nil. assumption. reflexivity.
}
destruct (encoder tail) as [bytes3 rest] eqn:encoder_tail.
specialize (IHtail bytes3 rest ltac:(reflexivity)).
destruct IHtail as [codes_tail [decode_bytes3 tail_eq]].
inversion encode_codes.
eapply utf8_spec_encoder_decoder_inverse_single in encoder_code; [| assumption |
apply decoder_spec].
rewrite dec_projects.
+ rewrite encoder_code.
+ rewrite decode_bytes3.
exists ([code] ++ codes_tail). split. reflexivity. inversion tail_eq. subst.
reflexivity.
+ apply decoder_spec.
+ apply (decoder_spec.(dec_input decoder)).
exists code. assumption.

```

**Qed.**

Por fim, é possível provar os teoremas de corretude:

**Theorem** `utf8_spec_encoder_decoder_inverse` : `forall` encoder decoder,  
`utf8_encoder_spec encoder ->`  
`utf8_decoder_spec decoder ->`  
`forall` codes bytes codes\_suffix,  
`encoder codes = (bytes, codes_suffix) ->`  
`exists` codes\_prefix, decoder bytes = (codes\_prefix, nil) /\ codes =  
`(codes_prefix ++ codes_suffix)%list.`

**Proof.**

```

intros encoder decoder encoder_spec decoder_spec.
induction codes as [| code tail]; intros bytes codes_suffix encode_codes.
- exists []. pose proof (enc_nil encoder encoder_spec). rewrite H in encode_codes.
inversion encode_codes.
split. apply dec_nil. assumption. reflexivity.
- replace (code :: tail) with ([code] ++ tail) in encode_codes by reflexivity.
rewrite enc_projects in encode_codes; [| assumption].
destruct (encoder [code]) as [bytes2 rest] eqn:encoder_code.
destruct rest.
2: {
  inversion encode_codes. subst.
  specialize (enc_output encoder encoder_spec code) as bytes_invalid.
  rewrite encoder_code in bytes_invalid. destruct bytes_invalid as [bytes_eq

```

```

rest_eq].
  inversion rest_eq. subst.
  exists nil. split. apply dec_nil. assumption. reflexivity.
}
destruct (encoder tail) as [bytes3 rest] eqn:encoder_tail.
specialize (IHtail bytes3 rest ltac:(reflexivity)).
destruct IHtail as [codes_tail [decode_bytes3 tail_eq]].
inversion encode_codes.
eapply utf8_spec_encoder_decoder_inverse_single in encoder_code; [ | assumption |
apply decoder_spec].
rewrite dec_projects.
+ rewrite encoder_code.
  rewrite decode_bytes3.
  exists ([code] ++ codes_tail). split. reflexivity. inversion tail_eq. subst.
reflexivity.
+ apply decoder_spec.
+ apply (decoder_spec.(dec_input decoder)).
  exists code. assumption.
Qed.

```

**Theorem** `utf8_spec_decoder_encoder_inverse_strong` : `forall` encoder decoder,

```

utf8_encoder_spec encoder ->
utf8_decoder_spec decoder ->
forall (codes_big codes: unicode_str) bytes bytes_suffix,
  ((length codes) <= (length codes_big))%nat ->
  decoder bytes = (codes, bytes_suffix) ->
  exists bytes_prefix, encoder codes = (bytes_prefix, nil) /\ bytes =
(bytes_prefix ++ bytes_suffix)%list.

```

**Proof.**

```

intros encoder decoder encoder_spec decoder_spec.
induction codes as [| code codes]; intros bytes bytes_suffix length decoder_bytes.
- exists []. split. apply enc_nil. assumption.
  apply dec_output in decoder_bytes as G; [|assumption].
  destruct G as [_ [prefix [decode_prefix [prefix_valid bytes_eq]]]].
  apply utf8_spec_decoder_nil_unique in decode_prefix; [|assumption].
  subst. reflexivity.
- replace (code :: codes) with ([code] ++ codes) in decoder_bytes |- * by
reflexivity.
  eapply utf8_spec_decoder_project_dual in decoder_bytes; [| apply encoder_spec |
assumption ].
  destruct decoder_bytes as [bytes1 [bytes2 [decoder_bytes1 [decoder_bytes2
bytes_eq]]]].
  eapply utf8_spec_decoder_encoder_inverse_single in decoder_bytes1; [| apply
encoder_spec | assumption].
  apply IHcodes in decoder_bytes2; [| simpl in length; lia].
  destruct decoder_bytes2 as [bytes_prefix [encoder_codes bytes2_eq]].
  rewrite enc_projects; [| assumption].
  rewrite decoder_bytes1. rewrite encoder_codes.
  exists (bytes1 ++ bytes_prefix).
  split. reflexivity. inversion bytes2_eq. inversion bytes_eq. subst.
  rewrite app_assoc. rewrite app_nil_r. reflexivity.

```

**Qed.**

**Theorem** `utf8_spec_decoder_encoder_inverse`: `forall` encoder decoder,

```

utf8_encoder_spec encoder ->
utf8_decoder_spec decoder ->

```

```

forall codes bytes bytes_suffix,
  decoder bytes = (codes, bytes_suffix) ->
  exists bytes_prefix, encoder codes = (bytes_prefix, nil) /\ bytes =
(bytes_prefix ++ bytes_suffix)%list.
Proof.
  intros encoder decoder encoder_spec decoder_spec codes bytes bytes_suffix.
  apply utf8_spec_decoder_encoder_inverse_strong with (codes_big := codes);
[assumption | assumption | lia].
Qed.

```

Vale ressaltar que para provar o caso do decodificador, é utilizada indução forte, visto que o passo indutivo não necessariamente é feita com apenas um byte por vez.

## 5 Implementação

Com a especificação feita, a implementação de um codificador e decodificador práticos é relativamente simples. Para implementar o codificador, primeiro é definida uma função que mapeia um codepoint numa sequência de bytes:

```

Definition utf8_encode_codepoint (n: codepoint) : @option (list byte) :=
  if (n <? 0) then
    None
  else if (n <=? 127) then
    Some [ n ]
  else if (n <=? 0x7ff) then
    let b1 := n / 64 in
    let b2 := n mod 64 in
    Some [ 192 + b1; 128 + b2 ]
  else if (andb (n <=? 0xffff) (orb (n <? 0xd800) (n >? 0xdfff))) then
    let r := n / 64 in
    let b1 := r / 64 in
    let b2 := r mod 64 in
    let b3 := n mod 64 in
    Some [ 224 + b1; 128 + b2; 128 + b3 ]
  else if (andb (n <=? 0x10ffff) (n >? 0xffff)) then
    let r1 := n / 64 in
    let r2 := r1 / 64 in
    let b1 := r2 / 64 in
    let b2 := r2 mod 64 in
    let b3 := r1 mod 64 in
    let b4 := n mod 64 in
    Some [ 240 + b1; 128 + b2; 128 + b3; 128 + b4 ]
  else
    None.

```

Assim, o codificador é definido como uma função que recursivamente mapeia o mapeamento acima, parando quando a lista acaba ou quando o mapeamento retorna `None`.

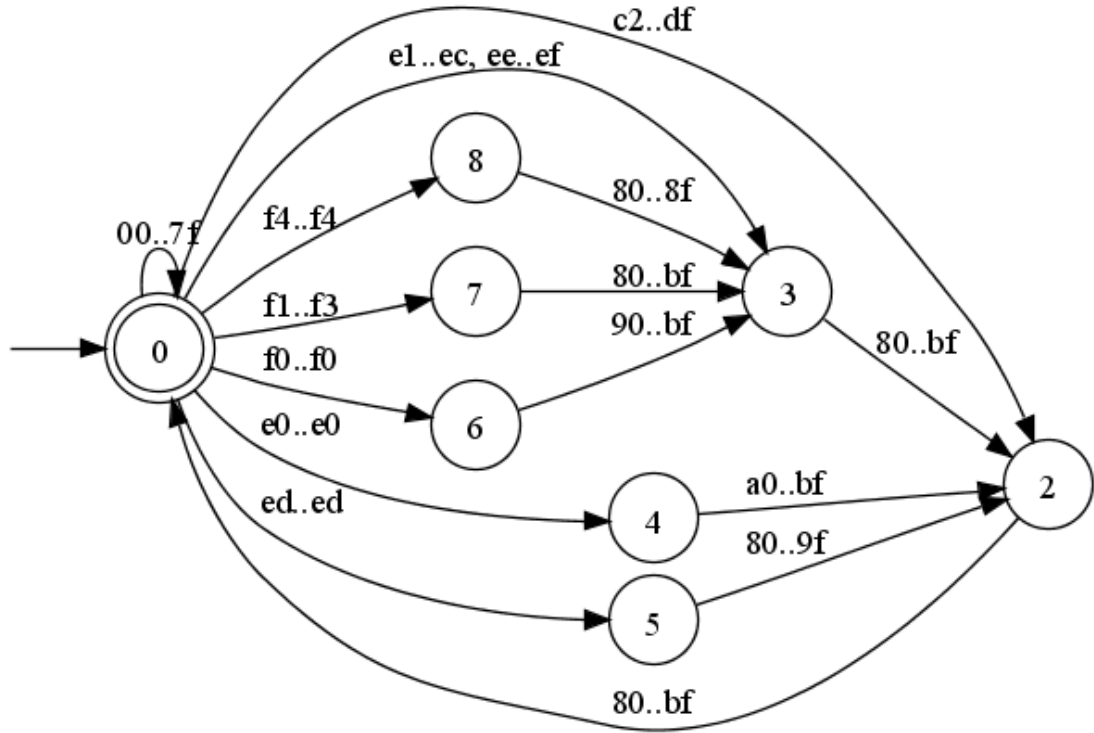
```

Fixpoint utf8_encode (unicode: unicode_str) : (list byte) * (list codepoint) :=
  match unicode with
  | [] => ([], [])
  | code :: unicode_rest =>
    match utf8_encode_codepoint code with
    | None => ([], code :: unicode_rest)
    | Some bytes =>
      let (bytes_rest, unicode_rest) := utf8_encode unicode_rest in
      (bytes ++ bytes_rest, unicode_rest)

```

end  
end.

Para implementar o decodificador, é utilizado um autômato de estado finito. Um DFA pode ser derivado observando cada linha da Tabela 5, e considerando quais caracteres podem ser lidos em cada parte.



A partir desse grafo, define-se o conjunto de possíveis estados, e uma enumeração de todos os possíveis estados úteis que aparecem no grafo:

```

Inductive parsing_state :=
  Initial
| Expecting_1_80_BF
| Expecting_2_80_BF
| Expecting_3_80_BF
| Expecting_2_80_9F
| Expecting_2_A0_BF
| Expecting_3_90_BF
| Expecting_3_80_8F.

```

```

Inductive byte_range :=
| Range_00_7F
| Range_80_8F
| Range_90_9F
| Range_A0_BF
| Range_C2_DF
| Byte_E0
| Range_E1_EC
| Byte_ED
| Range_EE_EF
| Byte_F0
| Range_F1_F3

```

| Byte\_F4

.

```

Definition byte_range_dec (b: byte) : option byte_range :=
  if b <? 0 then
    None
  else if b <=? 0x7f then
    Some Range_00_7F
  else if b <=? 0x8f then
    Some Range_80_8F
  else if b <=? 0x9f then
    Some Range_90_9F
  else if b <=? 0xbf then
    Some Range_A0_BF
  else if b <=? 0xc1 then
    None
  else if b <=? 0xdf then
    Some Range_C2_DF
  else if b ==? 0xe0 then
    Some Byte_E0
  else if b <=? 0xec then
    Some Range_E1_EC
  else if b ==? 0xed then
    Some Byte_ED
  else if b <=? 0xef then
    Some Range_EE_EF
  else if b ==? 0xf0 then
    Some Byte_F0
  else if b <=? 0xf3 then
    Some Range_F1_F3
  else if b ==? 0xf4 then
    Some Byte_F4
  else
    None.

```

Também são definidas funções auxiliares para representar as operações de extração de bits relevantes.

```

Definition push_bottom_bits (carry: codepoint) (b: byte): codepoint :=
  carry * 64 + (b mod 64).

```

```

Definition extract_7_bits (b: byte) : codepoint :=
  b mod 128.

```

```

Definition extract_5_bits (b: byte) : codepoint :=
  b mod 32.

```

```

Definition extract_4_bits (b: byte) : codepoint :=
  b mod 16.

```

```

Definition extract_3_bits (b: byte) : codepoint :=
  b mod 8.

```

Por fim, é definida a função `next_state`, que calcula o próximo estado do DFA a partir do estado atual e do byte visto. Para representar o fim de um codepoint, é criado o tipo `parsing_result`:

```

Inductive parsing_result :=
  Finished (codep: codepoint)
| More (state: parsing_state) (acc: codepoint).

```

```

Definition next_state (state: parsing_state) (carry: codepoint) (b: byte) : @option
parsing_result :=
  match (state, byte_range_dec b) with
  | (Initial, Some Range_00_7F) => Some (Finished (extract_7_bits b))
  | (Initial, Some Range_C2_DF) => Some (More Expecting_1_80_BF (extract_5_bits b))
  | (Initial, Some Byte_E0)      => Some (More Expecting_2_A0_BF (extract_4_bits b))
  | (Initial, Some Range_E1_EC)
  | (Initial, Some Range_EE_EF) => Some (More Expecting_2_80_BF (extract_4_bits b))
  | (Initial, Some Byte_ED)      => Some (More Expecting_2_80_9F (extract_4_bits b))
  | (Initial, Some Byte_F0)      => Some (More Expecting_3_90_BF (extract_3_bits b))
  | (Initial, Some Range_F1_F3) => Some (More Expecting_3_80_BF (extract_3_bits b))
  | (Initial, Some Byte_F4)      => Some (More Expecting_3_80_8F (extract_3_bits b))
  | (Initial, _) => None
  | (Expecting_1_80_BF, Some Range_A0_BF)
  | (Expecting_1_80_BF, Some Range_90_9F)
  | (Expecting_1_80_BF, Some Range_80_8F) => Some (Finished (push_bottom_bits carry
b))
  | (Expecting_2_80_BF, Some Range_80_8F)
  | (Expecting_2_80_BF, Some Range_90_9F)
  | (Expecting_2_80_9F, Some Range_80_8F)
  | (Expecting_2_80_9F, Some Range_90_9F)
  | (Expecting_2_80_BF, Some Range_A0_BF) => Some (More Expecting_1_80_BF
(push_bottom_bits carry b))
  | (Expecting_3_80_BF, Some Range_80_8F)
  | (Expecting_3_80_BF, Some Range_90_9F)
  | (Expecting_3_80_BF, Some Range_A0_BF)
  | (Expecting_3_90_BF, Some Range_90_9F)
  | (Expecting_3_90_BF, Some Range_A0_BF)
  | (Expecting_3_80_8F, Some Range_80_8F) => Some (More Expecting_2_80_BF
(push_bottom_bits carry b))
  | (Expecting_2_A0_BF, Some Range_A0_BF) => Some (More Expecting_1_80_BF
(push_bottom_bits carry b))
  | (Expecting_3_80_8F, Some Range_90_9F)
  | (Expecting_3_80_8F, Some Range_A0_BF) => None
  | _ => None
  end.

```

A função do decodificador, então, é uma função que recursivamente calcula o próximo estado utilizando `next_state`. Quando o resultado é um codepoint finalizado, a função volta para o estado inicial e começa a ler um novo codepoint.

```

Fixpoint utf8_dfa_decode_rec (bytes: list byte) (carry: codepoint) (state:
parsing_state) (consumed: list byte)
: unicode_str * (list byte) :=
  match bytes with
  | nil => ([], consumed)
  | cons b rest =>
    match next_state state carry b with
    | Some (Finished codep) =>
      let (vals, rest) := utf8_dfa_decode_rec rest 0x00 Initial [] in
      (codep :: vals, rest)
    | Some (More state codep) =>
      utf8_dfa_decode_rec rest codep state (consumed ++ [b])

```



```

      | None => ([], consumed ++ bytes)
    end
  end.

```

**Definition** `utf8_dfa_decode` (bytes: `list byte`) : `unicode_str * (list byte) :=`  
`utf8_dfa_decode_rec bytes 0x00 Initial []`.

Note que, pelas restrições de ser um *parser* parcial, é necessário guardar os bytes consumidos equivalentes ao codepoint atual, de modo a não jogar fora bytes se apenas um da sequência for inválido.

## 5.1 Corretude da implementação

### Bibliografia

DELAWARE, B. et al. Narcissus: correct-by-construction derivation of decoders and encoders from binary formats. **Proceedings of the ACM on Programming Languages**, v. 3, n. ICFP, p. 1–29, jul. 2019.

GEEST, M. VAN; SWIERSTRA, W. **Generic packet descriptions: verified parsing and pretty printing of low-level data**. Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development. **Anais...: ICFP '17**. ACM, set. 2017. Disponível em: <<http://dx.doi.org/10.1145/3122975.3122979>>

KOPROWSKI, A.; BINSZTOK, H. TRX: A Formally Verified Parser Interpreter. Em: **Programming Languages and Systems**. [s.l.] Springer Berlin Heidelberg, 2010. p. 345–365.

RAMANANANDRO, T. et al. **Secure Parsing and Serializing with Separation Logic Applied to CBOR, CDDL, and COSE**. Disponível em: <<https://arxiv.org/abs/2505.17335>>.

SENJAK, C.-S.; HOFMANN, M. **An implementation of Deflate in Coq**. Disponível em: <<https://arxiv.org/abs/1609.01220>>.

THÉRY, L. **Formalising Huffman's algorithm**. [s.l.: s.n.]. Disponível em: <<https://hal.science/hal-02149909>>.

THE UNICODE CONSORTIUM. **The Unicode Standard, Version 17.0.0**. South San Francisco, CA: The Unicode Consortium, 2025.

W3TECHS. **w3techs.com Usage statistics of UTF-8 for websites**. , 2025.

YE, Q.; DELAWARE, B. **A verified protocol buffer compiler**. Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs. **Anais...: CPP '19**. ACM, jan. 2019. Disponível em: <<http://dx.doi.org/10.1145/3293880.3294105>>

YERGEAU, F. **UTF-8, a transformation format of ISO 10646**. Disponível em: <<https://www.rfc-editor.org/info/rfc3629>>.