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. Três formatos para serializar *codepoints* em bytes existem, UTF-8, UTF-16 e UTF-32; entretanto, o formato mais ubíquito é UTF-8, pela sua retro compatibilidade com ASCII, e a capacidade de economizar bytes. Apesar disso, vários problemas aparecem ao implementar um programa codificador e decodificador de UTF-8 semânticamente correto, e inúmeras vulnerabilidades estão associadas a esse processo. Neste trabalho será utilizada verificação formal através de provadores de teoremas interativos, não apenas para enumerar todas as propriedades dadas na especificação do UTF-8, mas principalmente para desenvolver implementações e provar que estão corretas. Primeiro, uma implementação simplificada será desenvolvida, focando em provar todas as propriedades, depois uma implementação focada em eficiência e performance será dada, junto com provas de que as duas são equivalentes. Por fim essa implementação será extraída para um programa executável, e sua performance será comparada com soluções existentes não verificadas formalmente.

Contents

1 Introdução 1
2 Trabalhos relacionados 2
2.1 Codificadores e decodificadores 3
3 Unicode 4
3.1 UCS-2 e UTF-16 5
3.2 UTF-8 6
4 Formalização 8
4.1 Parsers monádicos 9
Bibliography 12

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.

Validação 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 da validação 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, de um modo geral, 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 desapercebidos 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 resultados, 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 de 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, 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 desse 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 Unicode fora escolhido pela sua difusão em praticamente todos os contextos e linguagens de programação. O padrão Unicode de representação de caracteres é ubíquito na comunicação na internet, e seu principal formato de codificação e decodificação, UTF-8, é utilizado em mais de 98% das web páginas (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. Para implementar isso corretamente, faz sentido utilizar ferramentas de verificação formal para assegurar a semântica desse programa, visto que ler e escrever UTF-8 é muito comum em diferentes partes da internet.

Assim, será utilizado o provador interativo Coq para desenvolver um programa que codifica e decodifica bytes no padrão UTF-8 e formalizar uma prova de que esse programa não aceita bytes inválidos de acordo com a especificação. Isso será feito através de dois programas complementares, um que aceita bytes e retorna *code points* – chamado de decodificador – e seu programa dual, que aceita *code points* e retorna *bytes* – o codificador. Além provar propriedades importantes sobre cada um dos programas, o cerne da corretude está na prova de que todo resultado válido de um dos programas é aceito corretamente pelo outro.

Performance e eficiência do programa final também serão consideradas, com o objetivo de mostrar que não é necessario descartar a eficiência para obter um programa correto utilizando essa metodologia. Para tal, uma implementação do codificador baseada em autômatos finitos será desenvolvida, em paralelo à prova de que esse é exatamente equivalente ao codificador correto.

2 Trabalhos relacionados

O compilador de C *CompCert* (LEROY (2006)) é um dos maiores expoentes da área da verificação formal, oferencedo um compilador da linguagem C com provas de corretude de funcionalidade. Ao desenvolver um modelo formal de memória contendo ponteiros, o compilador permite não somente descrever transformações entre programas, mas também provar matematicamente que tais transformações preservam a semântica do programa original. Dessa forma, torna-se trivial introduzir otimizações que simplificam o programa, garantindo que essas não introduzem *bugs*. De fato, YANG et al. (2011) mostraram que, dentre todos os compiladores de C tradicionais, incluindo `gcc` e `clang`, o *CompCert* fora o único a não apresentar sequer um *bug* de compilação após testes minuciosos.

O microkernel *seL4* (KLEIN et al. (2014)) é o primeiro e único microkernel de propósito geral que possue uma prova de corretude de funcionalidade, isto é, que seu código em *C* implementa o modelo abstrato de sua especificação, o que o torna livre de problemas de *stack overflow*, *deadlocks*, erros de aritmética e outros problemas de implementação. Além disso, provas de garantia de disponibilidade, integridade, e performance no pior caso são todas formalmente verificadas, fazendo com que esse micro-kernel tenha ampla adoção em casos de uso de missão crítica.

O algoritmo de consenso *Raft* é um exemplo de algoritmo implementado utilizando bases de verificação formal, sendo desenvolvido com base na lógica de separação, oferencendo garantias de

segurança, disponibilidade e corretude, sendo adotado por inúmeras implementações de bancos de dados, como *ScyllaDB*, *MongoDB*, *ClickHouse*, *Apache Kafka*, e muitos outros.

Além de útil para produção de software e algoritmos corretos, as ferramentas de verificação formal são ótimos fundamentos para o desenvolvimento da matemática. GONTHIER (2008) completamente formalizaram a prova do teorema das 4 cores no provador interativo Coq, cuja prova inicial, por APPEL; HAKEN (1977), era extremamente complicada e sofria críticas por sua complexidade. GONTHIER et al. (2013) formalizaram o teorema de Feit-Thompson, cuja prova manual contém mais de 10 mil páginas. Mais recentemente, em 2024, o valor de BusyBeaver(5) fora calculado como 47.176.870 utilizando Coq (BBCHALLENGE (2024)), cujo processo consiste em decidir dentre todas as máquinas de Turing de tamanho 5 que terminam, a que leva o maior número de passos para terminar.

2.1 Codificadores e decodificadores

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 táticas – uma espécie de macro para gerar provas – 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.

KOPROWSKI; BINSZTOK (2010) forneceram uma implementação similar para linguagens que podem ser descritas por PEGs em Coq, 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 abitrários, focando no caso de uso dos padrões ASN.1, fornecendo uma formalização de formato IPV4.

YE; DELAWARE (2019) descrevem o processo de implementar em Coq um gerador do par codificador/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 decodicam os formatos específicos do arquivo, junto das provas de que os programas gerados devem obedecer a essa semântica corretamente.

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.

O sistema de verificação formal não foi usado apenas nesse contexto, e é possível encontrar formalizações de algoritmos mais complexos. Laurent Théry formalizou uma implementação do algoritmo de Huffman, frequentemente utilizado em padrões de compressão sem perda de dados. Similarmente, SENJAK; HOFMANN (2016) construiram uma implementação completa do algoritmo de Deflate, usado em formatos como PNG e GZIP.

Apesar de não fornecerem provas de corretude, KEISER; LEMIRE (2020) desenvolveram um algoritmo para validar UTF-8 utilizando SIMD, 10 vezes mais rápido do que outros algoritmos da época. Como implementações padrões de validações de UTF-8, temos também

as bibliotecas padrões de Rust e Swift, cuja principal implementação de `String` sempre valida que seus bytes são UTF-8 (deveria citar isso aqui?).

3 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. Unicode é o padrão mais utilizado hoje em dia por permitir codificar caracteres de praticamente todas as linguagens existentes de modo integrado, removendo a necessidade de utilizar outros sistemas de codificação. Para entender seu design e funcionamento, faz-se necessário entender como funcionavam os seus antecessores.

> Definição: ***code point*** 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. No caso em que o número é menor ou igual a `65535` (`xFFFF`), é tradicional omitir os zeros mais significativos.

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

x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS						SO	SI
DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

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 último bit previamente ignorado. Esses foram criandos primariamente para adicionar suporte à caracteres especificos 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, conhecido como *code points*, 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 (≤ 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 extremamente 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 fora 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 extremamente ambicioso, esse sistema rapidamente ganhou adoção mundial, massivamente simplificando a comunicação na internet.

3.1 UCS-2 e UTF-16

Em 1991, a versão 1.0 do Unicode fora 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 as linguagens Java (1995) e

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 *code points* distintos (32 bits) representem um único *code point*, isto é, pares de caracteres, denominados *surrogate pairs*, representando um único caractere. 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 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.

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`	`wwwwxxxx yyyyzzzz`	16 bits
`U+10000`	`U+10FFFF`	`110110vv vvwwwwxx 110111xx yyyyzzzz`	20 bits

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:

Inicio..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`	$2^{20} + 2^{16} - 2^{11}$	*Code points* representáveis

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 divido 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 (python e APACHE, respectivamente, mais detalhes).

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), bem como para formatos como 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.

3.2 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 para *code points* no BMP, 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 começa sempre com `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, wwww, xxxx, yyyy, zzzz]. Utilizando essa notação, a serialização deste pode ser vista como:

Início	Fim	Bytes	Bits relevantes
U+0000	U+007F	0yyyzzzz	7 bits
U+0080	U+07FF	110xxxyy 10yyzzzz	11 bits
U+0800	U+FFFF	1110wwww 10xxxxyy 10yyzzzz	16 bits
U+10000	U+10FFFF	11110uvv 10vvwwww 10xxxxyy 10yyzzzz	21 bits

É importante notar que os primeiros 127 *code points* são representados exatamente igual caracteres ASCII (e sistemas extendidos), algo extremamente desejável não apenas para retro 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+1FFFFF`), 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

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`).

Portanto, para escrever um programa que codifica e decodifica UTF-8 corretamente, precisamos mostrar que esse programa sempre respeita essas propriedades.

4 Formalização

Como dito anteriormente, a formalização é baseada na implementação de um codificador e decodificador de forma conjunta, já que representam a ação de desfazer a ação do outro. Enquanto o codificador recebe uma sequência de *code points* e retorna uma sequência de bytes ou um erro, o decodificador recebe uma sequência de bytes e retorna uma sequência de *code points* ou um erro. O cerne da corretude, e o motivo de desenvolver o projeto dessa forma, está em mostrar formalmente que a seguinte propriedade vale para os programas: uma sequência de bytes `b` é aceita pelo decodificador como a sequência de *code points* `c`, se, e somente se, o codificador retornar bytes `b` ao receber *code points* `c`.

Além disso, é necessário mostrar que bytes só são aceitos pelo decodificador quando *code points* resultantes são válidos, e da mesma forma, que todo *code point* aceito pelo codificador é válidos. É importante ressaltar que a noção de validade de *code point* é a descrita anteriormente: um *code point* é **UTF-8 válido** quando o número representado é menor do que `U+10FFFF` e não está no intervalo `U+DB00..U+DFFF`. Da mesma forma, uma sequência de bytes é tida como **UTF-8 válida** se essa representa a serialização de um *code point* **válido**, e é a menor possível para o respectivo intervalo do *code point*.

Assim, a corretude geral é baseada nas seguintes 3 propriedades principais:

1. Todo *code point* **válido** possui uma sequência de bytes **válida** correspondente.
2. Toda sequência de bytes **válida** possui um *code point* **válido** correspondente.
3. Toda sequência de bytes **válida** é a menor possível para o intervalo correspondente.
4. Para tanto o codificador quanto o decodificador, se dado o valor `v` a implementação retorna corretamente uma sequência `s`, a outra sempre é capaz de converter essa sequência `s` no valor `v`.

Dessa forma, há de se mostrar como implementar esses programas em Coq, bem como traduzir tais propriedades para código checável por máquina.

4.1 Parsers monádicos

Por simplicidade, a representação de ambos codificadores e decodificadores é dada pela mesma mônade, `result`, definida em Coq como:

```
Inductive result {T E: Type} : Type :=
| Ok (x: T) : @result T E
| Err (err: E) : @result T E.
```

Isso é, `result` é um tipo que possui dois construtores, `Ok` e `Err`, que representam duas possibilidades de resultado para uma função. Utilizando-a, pode-se definir a seguinte função:

```
Definition bind {A B E} (r: @result A E) (f: A -> @result B E) :=
  match r with
  | Ok p => f p
  | Err err => Err err
  end.
```

A função `bind` permite construir uma continuação de um resultado caso ele seja `Ok`, recebendo uma função `f` que é chamada no valor interno `p`. Essa função é tão importante que possue uma notação especial, `let*`, que permite aplicá-la sucessivamente considerando que o resultado anterior seja `Ok`. Isto é, utilizar `bind` em cadeia permite construir computações que só acontecem caso a computação anterior tenha sido sucesso, e que imediatamente retornam `Err` caso qualquer computação intermediária falhe.

```
Notation "'let*' p ':=' c1 'in' c2" :=
 (bind c1 (fun p => c2))
    (at level 61, p as pattern, c1 at next level, right associativity).
```

Utilizando `result`, o tipo `parser` é definido da seguinte forma:

```
Definition parser (T: Type) {C E: Type} := list C -> @result (T * (list C)) E.
```

Isto é, um `parser` é uma função que recebe uma lista de elementos de tipo `C` – que pode ser interpretado como o tipo do "caractere" que o `parser` lê – e retorna um `result`: no caso de sucesso, retorna um par de um valor de tipo `T` e o resto que não fora lido de tipo `list C`, e no caso de erro, retornasse um valor de tipo `E`.

Um parser pode ser visto como uma leitor parcial, que no caso de sucesso retorna um valor `T` lido junto do resto que não fora processado, de forma que parsers possam ser encadeados sequencialmente aplicando um novo `parser` ao resto não lido pelo anterior.

Para facilitar o encadeamento, são definidas funções – chamadas de combinadores – que constroem parsers a partir de outros parsers, de modo a oferecer blocos componíveis. Por exemplo, a seguinte função `maybe` recebe um `parser` que retorna um valor `A` e retorna um `parser` que retorna o valor `option A`:

```
Definition maybe {A I E} (p: @parser A I E) : @parser (option A) I E :=
  fun s =>
    match p s with
    | Ok (x, rest) => Ok (Some x, rest)
    | Err _        => Ok (None, s)
    end.
```

Da mesma forma, pode-se utilizar a notação `let*` para definir `parser_map`, que recebe um `parser A` e uma função `f: A -> B` e retorna um `parser B` simplesmente aplicando essa função ao resultado da computação:

```
Definition parser_map {A B I E} (f: A -> B) (p: @parser A I E) : @parser B I E :=
  fun s =>
    let* (x, rest) := p s in
    Ok (f x, rest).
```

Outro combinador importante é `predicate`, que recebe uma função predicado `pred: I -> bool` e retorna `Ok` caso o primeiro caractere da entrada satisfaz o predicado:

```
Definition predicate {I E} (pred: I -> bool) (err: option I -> E) : @parser I I E :=
  fun s =>
    match s with
    | [] => Err (err None)
    | c :: rest =>
        match pred c with
        | true => Ok (c, rest)
        | false => Err (err (Some c))
        end
    end.
```

Por fim, o último combinador utilizado na biblioteca é `all`, que recebe um `p: parser A` e retorna `Ok` se todos caracteres são consumidos por sucessivas aplicações de `p`.

```
Fixpoint all_aux { A I E} (p: @parser A I E) (fuel: nat) : @parser (list A) I E :=
  fun s =>
    match fuel with
    | 0 => Ok ([], s)
    | S fuel' =>
        match s with
        | [] => Ok ([], s)
        | _ =>
            match p s with
            | Err e => Err e
            | Ok (val, rest) =>
                let* (vals, rest) := all_aux p fuel' rest in
                Ok (val :: vals, rest)
            end
        end
    end.

Definition all {A I E} (p: @parser A I E): @parser (list A) I E :=
  fun s => all_aux p (S (length s)) s.
```

Em Coq, fortes limitações são colocadas à funções recursivas, visto que é necessário mostrar que toda função deve terminar necessariamente, pois permitir funções que rodem infinitamente introduzem inconsistências à lógica interna. Isto é, de uma função que roda infinitamente é possível construir uma prova para qualquer proposição `P`, incluindo o tipo `⊥` que não possui nenhum construtor.

Para demonstrar terminação, normalmente é necessário mostrar que o passo recursivo é aplicado a um elemento estritamente menor do que o passo anterior, de modo que não haja como a sequência continuar infinitamente. Entretanto, como o passo recursivo de `all_aux` é dado diretamente no resultado de `p s`, o sistema de terminação não é capaz de mostrar que `rest` deve ser menor do que `s`, e portanto recusa a definição trivial. Assim, é introduzido uma variável "gás", de forma que a função sempre utilize-a no passo recursivo. No caso em que `fuel == 0`, a função imediatamente retorna.

Idealmente, a função `all` nunca atingiria o caso `fuel == 0`, mas infelizmente não é possível provar que isso é verdade, visto que o `parser` não possui restrições sobre o que é retornado no resto. De fato, um `parser` malicioso que sempre retorna a mesma lista constante como resto fará com que `all` entre em loop infinito sem a limitação do `gás`.

Entretanto, é possível concluir logicamente que se um parser `p` sempre consome algum caracter do resto ao retornar sucesso, e existem `length s` caracteres numa lista `s`, então `all_aux p` com gás `(length s) + 1` nunca atingirá o caso `fuel == 0`.

Essa lógica acima pode ser formalizada e provada em Coq como um teorema importante para a formalização da implementação, denominado de `all_aux_saturation_aux`:

```
Theorem all_aux_saturation_aux : forall A I E processor,
  (forall suffix response text,
    processor text = Ok (response, suffix) ->
    length suffix < length text) ->
  forall n text fuel,
  (S (length text)) < n ->
  (S (length text)) <= fuel ->
  @all_aux A I E processor (S (length text)) text = @all_aux A I E processor fuel
text.
Proof.
  intros A I E processor processor_good.
  induction n; intros text fuel.
  - intros H. inversion H.
  - intros text_bounded enough_fuel.
    destruct text as [|text_head text_tail].
    destruct fuel; try inversion enough_fuel. reflexivity. simpl. destruct (processor
[]) eqn:response_definition.
    destruct x as [response suffix].
    exfalso. assert (@length I suffix < @length I nil).
    apply processor_good with (response := response) (suffix := suffix).
    apply response_definition.
    inversion H1.
    reflexivity. simpl in text_bounded.
    destruct fuel. exfalso. inversion enough_fuel.
    simpl. destruct (processor (text_head :: text_tail)) eqn:response_definition.
    + destruct x as [val rest].
      replace (all_aux processor fuel rest) with (all_aux processor (S (length
text_tail)) rest).
      reflexivity.
      assert (length rest < length (text_head :: text_tail)). {
        apply processor_good with (response := val).
        apply response_definition.
      } {
      replace
        (all_aux processor (S (length text_tail)) rest)
      with
        (all_aux processor (S (length rest)) rest).
      apply IHn. simpl in H. lia. lia.
      apply IHn. simpl in H. lia.
      simpl in H. lia.
      }
    + reflexivity.
Qed.
```

A prova pode ser interpretada da seguinte forma: dado um `parser` denominado `processor`, para qual vale a propriedade de que se `processor text` retorna `Ok` então o tamanho do resto do resultado é estritamente menor do que o tamanho da entrada, podemos provar que para qualquer valor de gás `n` maior ou igual a `length text + 1`, o resultado final deve ser igual a aplicar o parser com o tamanho `length text + 1`. A prova utiliza a variável `n` para realizar uma indução forte, de forma que a hipótese indutiva possa ser aplicada a qualquer `fuel < n`.

Bibliography

APPEL, K.; HAKEN, W. Every planar map is four colorable. Part I: Discharging. **Illinois Journal of Mathematics**, v. 21, n. 3, Sep. 1977.

BBCHALLENGE. **We have proved "BB(5) = 47,176,870"**. , 2024.

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, Sep. 2017. Disponível em: <http://dx.doi.org/10.1145/3122975.3122979>

GONTHIER, G. **The Four Colour Theorem: Engineering of a Formal Proof**. (D. Kapur, Ed.)Computer Mathematics. **Anais**...Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

GONTHIER, G. et al. A Machine-Checked Proof of the Odd Order Theorem. In: **Interactive Theorem Proving**. [s.l.] Springer Berlin Heidelberg, 2013. p. 163–179.

KEISER, J.; LEMIRE, D. Validating UTF-8 In Less Than One Instruction Per Byte. 2020.

KLEIN, G. et al. Comprehensive formal verification of an OS microkernel. **ACM Transactions on Computer Systems**, v. 32, n. 1, p. 1–70, Feb. 2014.

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

LEROY, X. **Formal certification of a compiler back-end, or: programming a compiler with a proof assistant**. 33rd ACM symposium on Principles of Programming Languages. **Anais**...ACM Press, 2006. Disponível em: <http://xavierleroy.org/publi/compiler-certif.pdf>

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>.

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

YANG, X. et al. Finding and understanding bugs in C compilers. **ACM SIGPLAN Notices**, v. 46, n. 6, p. 283–294, Jun. 2011.

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>