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 isso. Neste trabalho será utilizada verificação formal através de tipos dependentes, não apenas para enumerar todas as propriedades dadas na especificação do UTF-8, mas principalmente para desenvolver implementações e provar que estas 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, e por fim essa implementação será extraída para um programa executável.

Contents

1 Introdução

O processo de desenvolvimento de software pode ser separado em dois problemas distintos: o de validação, que pretende assegurar que o programa a ser desenvolvido resolve um problema desejado, e o de verificação, que assegura que o programa desenvolvido implementa as especificações formadas na fase de validação.

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 (QA) 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, visto que pela natureza dos testes, não há como checarem com totalidade que o programa está correto, visto que teriam de ter um número impraticável de casos – muitas vezes infinito. Por isso, é cotidiano que erros e *bugs* passem desapercebidos por baterias gigantescas de testes, especialmente quando ocorrem em combinações muito específicas da entrada, já que não conseguem cobrir todos os possíveis casos.

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.

O compilador de C *CompCert* (LEROY (2006)) é um dos maiores expoentes dessa área. 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.

Assim, este trabalho tem o objetivo de utilizar a linguagem 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 *codepoints* (decodificador), e seu programa dual, que aceita *codepoints* e retorna *bytes* (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 considerados, com o objetivo de mostrar que não é necessario descartar a eficiência para obter um programa correto. Para tal, uma implementação do codificador baseada em autômatos finitos será desenvolvida, bem como uma prova de que esse é exatamente equivalente ao codificador correto.

2 Trabalhos relacionados

3 Unicode

Sistemas antigos de codificação de caracteres tradicionalmente utilizam um mapeamento um pra um entre caracteres e bytes, tanto por simplicidade de codificação quanto por eficiência de armazenamento de memória. Programas que decodificam bytes em caracteres nesses sistemas antigos são extremamente simples, e podem ser resumidos a tabelas de conversão direta, conhecidas como *code pages*.

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 outras famosas. 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

Para entender a razão por trás do formato moderno do Unicode, faz-se necessário entender a história por trás de sua criação. Em 1991, a versão 1.0 do Unicode fora lançado como uma codificação de tamanho fixo de 16 bits, chamada de UCS-2 – *Universal Coding System* – capaz de representar 65536 caracteres, chamados de *code points*, das mais diversas línguas.

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 possível. Assim, para estendê-lo mantendo-o retro compatível, decidiram reservar parte da tabela para que dois *code points* distintos (32 bits) representem um único *code point*, isto é, pares de *code units*, 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* é o padrão conhecido como UTF-16, que rapidamente obteve adoção de sistemas de grande relevância, como as linguagens Java e JavaScript, o sistema de UI Qt e até mesmo as APIs do Windows.

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 vv wwwwxx 110111xx xxyyyyzz`	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 byte 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` $\setminus$ `U+D800..U+DFFF`	$2^{20} + 2^{16} - 2^{11}$	Codepoints representáveis

Disso, pode-se inferir que um *code point* **válido** é um número de 21 bits que:
1. Não está no intervalo `D800..DFFF`.
2. Não ultrapassa `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 que utilizam ASCII (1 byte por caractere), bem como para formatos como HTML e JSON utilizados na internet, que usam muitos caracteres nos primeiros 127 *code points* – `<`, `>`, `{`, `:`. 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 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

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.

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.

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.

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>

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