Um Projeto de Bloco para Funcional Introdutório Programação em Haskell

Matthew Poole
School of Computing
University of Portsmouth, Reino
Unido matthew.poole@port.ac.uk

Resumo—Este artigo descreve o projeto visual de blocos para edição de código na linguagem funcional Haskell. O objetivo do ambiente baseado em blocos proposto é apoiar os passos iniciais dos alunos no aprendizado da programação funcional. Blocos de expressão e slots são moldados para garantir que o código construído seja sintaticamente correto e preserve o uso convencional de espaço em branco. O projeto visa ajudar os alunos a aprender o sofisticado sistema de tipos de Haskell, que muitas vezes é considerado um desafio para os programadores funcionais novatos. Os tipos são representados usando texto, cor e forma, e slots vazios indicam tipos de argumentos válidos para garantir que o código construído seja bem digitado.

I. INTRODUÇÃO

Ambientes baseados em blocos, como Scratch [1] e Snap! [2] oferecem várias vantagens sobre as linguagens tradicionais baseadas em texto para programadores novatos. Tais sistemas tendem a enfatizar a programação imperativa usando linguagens personalizadas. Existe algum trabalho sobre edição baseada em blocos para linguagens imperativas tradicionais, como Python [3], [4] e Grace [5], que visa apoiar os alunos aprendendo programação em ambientes educacionais formais.

Muitos alunos, principalmente aqueles que optam por se especializar em computação no ensino superior, enfrentam desafios adicionais ao aprender uma segunda ou terceira linguagem ou paradigma de programação. Vários educadores acreditam que aprender programação funcional é um passo importante no desenvolvimento de cientistas da computação e engenheiros de software [6], [7]. O conhecimento de programação funcional pode ser visto como uma habilidade prática importante dado o crescente número de linguagens que enfatizam a abordagem funcional (eg, Clojure, F#, Scala) ou que incluem construções funcionais (eg, Python, JavaScript, Java).

Haskell é frequentemente considerada uma boa linguagem para aprender programação funcional devido à sua sintaxe limpa e pureza funcional. Mas Haskell não é uma linguagem simples; seu sistema de tipos (particularmente seu conceito de classes de tipos) é bastante complexo, e as mensagens de erro do compilador (por exemplo, as do popular Compilador Haskell de Glasgow) costumam ser muito difíceis de compreender para iniciantes. Combinados com os esforços necessários para aprender um novo paradigma, esses problemas podem levar os alunos a dificuldades. O Helium [8], um subconjunto do Haskell para aprendizado de programação funcional, foi projetado para aliviar esses problemas. O lium não inclui classes de tipo e está focado em fornecer dicas, avisos e mensagens de erro aprimoradas para o aluno.

Em vez de definir uma linguagem ou compilador simplificado, a abordagem adotada aqui é expor o sistema de tipos de Haskell para o aluno, e para evitar erros de sintaxe e de tipo totalmente por meio da construção de programas baseados em blocos.

Existe algum trabalho recente na representação de tipos funcionais em ambientes baseados em blocos. TypeBlocks [9] inclui três formatos básicos de conectores (para listas, tuplas e funções) que podem ser combinados de qualquer maneira e em qualquer profundidade. O editor de blocos protótipo para Bootstrap [10], [11] representa cada um dos cinco tipos de uma linguagem funcional simples usando uma cor diferente, com uma cor neutra (cinza) usada para blocos polimórficos; os blocos cinza mudam de cor assim que seu tipo é determinado durante a construção do programa. Alguns recursos funcionais também foram adicionados ao Snap! [12] e para uma versão modificada do App Inventor [13].

Este artigo está estruturado da seguinte forma. Na próxima seção damos uma breve visão geral do núcleo da linguagem Haskell relevante para as ideias de projeto de blocos, que são então apresentadas na Seção III. A Seção IV conclui o artigo e discute trabalhos futuros.

II. VISÃO GERAL DE HASKELL

Haskell é uma linguagem funcional pura tipada estaticamente (não há estado nem efeitos colaterais, e as funções podem ser passadas e retornadas de outras funções). Haskell também apresenta inferência de tipo; uma das consequências disso é que os programadores raramente precisam atribuir explicitamente tipos às suas definições de função (e não consideraremos declarações de tipo aqui).

Haskell suporta polimorfismo (paramétrico): expressões (incluindo funções) podem ter mais de um tipo. Usando variáveis de tipo (normalmente a, b, c, . . .) para representar tipos polimórficos, declaramos o tipo (digamos) da função de identidade id como a -> a (escrito id :: a -> a). Aqui, a pode representar qualquer tipo, então id pode ser usado, por exemplo, em um contexto que requeira Ints (para dar Int -> Int) ou Chars (para dar Char -> Char).

As funções em Haskell são atualizadas; todas as funções são consideradas como tendo um único argumento. Uma função f :: a -> (b -> c), mais convencionalmente escrita f :: a -> b -> c, recebe um único argumento do tipo a e retorna uma função do tipo b -> c. As funções Curried podem ser parcialmente aplicadas; para x :: a, a aplicação parcial fx é do tipo b -> c. Um aplicativo 'completo' é escrito fxy (onde y :: b) e é do tipo c.

Haskell suporta sobrecarga (ou polimorfismo ad-hoc) através do uso de classes de tipo. Uma classe de tipo em Haskell pode ser vista como um conjunto de tipos que compartilham certas operações;

as operações suportadas por uma classe de tipo são conhecidas como seus métodos. Um tipo pode ser declarado como sendo uma instância de uma classe de tipo; uma declaração de instância inclui definições específicas de tipo (sobrecarregadas) de cada um dos métodos da classe de tipo.
A biblioteca padrão Haskell (o Prelude) inclui definições de muitas classes e instâncias de tipos. Por exemplo, a classe de tipo Num inclui os operadores numéricos '+', '-' e '*' como seus métodos. Os tipos de cada um desses operadores são Num a => a -> a -> a, que pode ser lido como: “para todo tipo a que é uma instância da classe Num, o operador tem tipo a -> a -> a .
O Prelude define todos os tipos numéricos (Int, Float, etc.) como instâncias da classe Num, cada um fornecendo definições sobrecarregadas desses três operadores; por exemplo, a instância Int que os define tem operações inteiras do tipo Int -> Int -> Int.

As classes de tipo podem ser relacionadas hierarquicamente, de modo que, para que um tipo seja declarado como uma instância de uma subclasse, ele também deve ser uma instância da superclasse. Por exemplo, a classe do tipo Fractional é uma subclasse de Num e inclui o método extra '/' (para divisão de ponto flutuante). Os tipos Float e Double são declarados como instâncias de Fractional, mas Int não.

III. DESENHO DE BLOCO

Os principais objetivos do projeto são representar totalmente todos os tipos de blocos de expressão e que o código construído seja formatado de acordo com a convenção.

A. Tipos básicos e polimórficos

Para expressões de tipos 'básicos' (tipos não parametrizados, não tuplos e não funcionais), rotulamos os blocos usando o nome de seu tipo junto com uma cor. A Fig. 1(a) fornece os rótulos para os tipos numéricos, de caracteres e booleanos de Haskell. Variáveis de tipo que representam rótulos; em qualquer tipo (polimórfico) não são usadas a, b, c, . . . em vez disso, eles são substituídos por padrões de hachura cinza, como os da Fig. 1(b). A intenção é que onde o mesmo padrão de hachura ocorre dentro de um conjunto de blocos conectados, ele representa qualquer tipo comum; diferentes padrões de hachura representam tipos potencialmente diferentes.

Fig. 1. Rótulos para (a) alguns dos tipos básicos e (b) polimórficos de Haskell.

B. Forma e comportamento do bloco

O elemento básico de uma definição de uma função f em Haskell é a equação que toma a forma f p1 . . . pn = r, onde cada pi é um padrão e r é o resultado. Algumas expressões (e, portanto, blocos) são sintaticamente válidas apenas dentro de padrões, outras apenas em resultados e algumas podem aparecer tanto em padrões quanto em resultados. Usamos diferentes formas de slot e bloco para impor a construção de código Haskell sintaticamente válido.

A Fig. 3 inclui uma equação para definir uma função f :: Shape -> Bool, que assume a existência de um tipo Shape. Para este artigo, não consideramos como a Forma é definida, como o tipo de f é determinado ou como as equações são criadas e combinadas; o foco aqui está na forma do vazio slots de padrão e resultado e os 'indicadores de tipo' dentro desses slots. Os cantos superiores angulares dos slots determinam quais variedades de blocos são sintaticamente legais e os indicadores determinam os tipos válidos; tanto a forma quanto o tipo do bloco precisam ser compatíveis com o slot para que uma gota seja aceita.

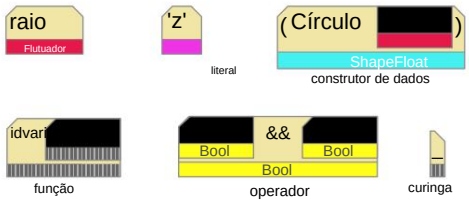

Fig. 2. Seis variedades de blocos com cantos angulares e rótulos de tipo que determinam onde eles podem ser soltos.

A Fig. 2 dá exemplos de blocos para seis variedades de expressões Haskell. Os três primeiros blocos são angulados em ambos os cantos superiores e podem ser colocados em qualquer slot (aqueles angulados à esquerda ou à direita e aqueles não angulados) se os indicadores de tipo corresponderem
Os blocos de função e operador não podem ser inseridos em slots inclinados à direita (ou seja, eles só podem ser usados em resultados) e o bloco curinga não pode ser inserido em nenhum slot inclinado à esquerda (só pode ser usado em padrões).

A Fig. 3 ilustra os efeitos de soltar blocos em slots.
O bloco radius é descartado legalmente no slot de argumento unangle do bloco Circle desde que os tipos coincidam.
Os blocos assumem as formas dos slots nos quais são soltos para mostrar em que contexto estão sendo usados e, assim, os ângulos desaparecem do bloco de raio. Quando o bloco Círculo é então colocado no slot esquerdo da equação, ele é remodelado para corresponder ao slot (agora está sendo usado como parte de um padrão). O slot de argumento (preenchido) do círculo é reformulado de maneira semelhante, assim como o bloco de raio que ele contém (já que todos os elementos de um padrão devem ser padrões). Se um bloco for removido de um slot, ele volta à sua forma original.

Observe que os parênteses são automaticamente adicionados e removidos conforme necessário. O bloco do construtor de dados Circle inclui parênteses inicialmente, pois são necessários na maioria dos contextos e fornecem espaço para o ângulo do bloco à direita; em alguns contextos (por exemplo, como resultado), os parênteses e o canto direito em ângulo desaparecem. Rotular os tipos na parte inferior dos blocos fará com que o código profundamente aninhado seja bastante alto; no entanto, isso não é considerado um problema importante, pois as funções geralmente são definidas usando poucas linhas de código.

Existem dois blocos polimórficos na Figura 2: a função de identidade id :: a -> a, e o curinga '_', que é usado em padrões para corresponder a qualquer valor e é considerado do tipo a.
A Fig. 4 ilustra o comportamento dos blocos polimórficos. Na Fig. 4(a), o bloco curinga é colocado no slot esquerdo (padrão) de uma equação; isso é válido, pois as formas de bloco e slot correspondem e a hachura cinza corresponde a qualquer tipo. O tipo do curinga muda para o do slot e vemos aqui que o nome do tipo é abreviado para preservar a formatação de código convencional.
Na Figura 4(b), o bloco 'z' é inserido no slot de argumento de id; como o argumento e o tipo de resultado são comuns (eles são hachurados da mesma maneira), ambos são recoloridos para corresponder ao tipo (Char) do bloco descartado. O bloco 'z' também muda de forma para corresponder ao slot que o contém (agora está sendo usado como parte de uma expressão de resultado).

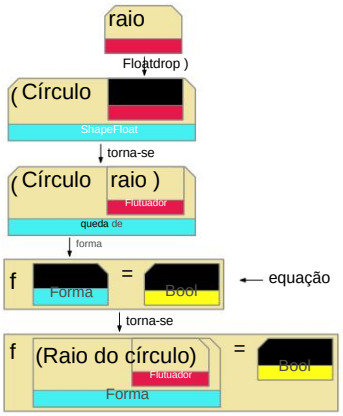

Fig. 3. Soltando blocos para formar o lado esquerdo (padrão) de uma equação.

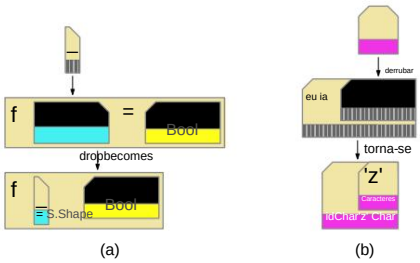

Fig. 4. Comportamento de blocos polimórficos: (a) um bloco curinga é descartado e redigitado para Shape; (b) um bloco Char se torna o argumento de id, que muda seu tipo para Char -> Char.

C. Tipos Parametrizados

Os tipos em Haskell podem ser parametrizados por um ou mais parâmetros de tipo. Um tipo parametrizado comumente usado é Maybe a, que representa valores opcionais; um valor do tipo Maybe a contém um valor do tipo a (representado pelo construtor de dados Just a) ou está vazio (Nothing). Por exemplo, a expressão Just 'z' é do tipo Maybe Char.
O tipo Both ab representa valores com um dos dois tipos possivelmente diferentes, usando construtores de dados Left a ou Right b. Por exemplo, a expressão Right 'z' é do tipoOU a Char.

A Fig. 5 mostra como os tipos parametrizados podem ser representados, usando cores para os próprios tipos e com áreas incorporadas acima e à direita para os tipos do(s) parâmetro(s).

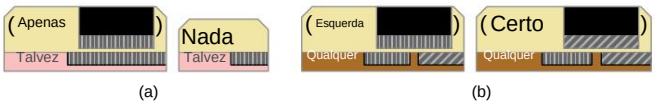

Fig. 5. Blocos para os construtores de dados dos tipos (a) Maybe a e (b) Ou um b.

D. Classes de tipo

Os rótulos para blocos polimórficos e slots cujo tipo é restrito por uma classe de tipo são coloridos em cinza e incluem o nome da classe de tipo envolvida. Eles são, portanto, diferenciados tanto dos tipos (que usam cores) quanto dos tipos polimórficos irrestritos (que usam hachura cinza).

A Fig. 6 ilustra os rótulos e comportamentos de blocos que envolvem classes de tipo. A expressão 3.14 em Haskell é do tipo Fractional a => a que é representada pelo texto Fractional em um fundo cinza. Observe que um espaço em branco extra foi adicionado ao redor do valor para permitir que o nome da classe de tipo seja fornecido por completo, pois não há necessidade de abreviar rótulos de tipo em blocos desconectados. Este bloco é colocado em um slot de argumento do bloco '*' (o operador '*' é do tipo Num a => a -> a -> a). Como Fractional é uma subclasse de Num, por meio de inferência de tipo, o bloco '*' é redigitado para Fractional a => a -> a -> a. Como Float é uma instância da classe de tipo Fractional, soltar ping do bloco radius no slot vazio restante é permitido e leva, novamente por meio de inferência de tipo, aos tipos de '*' e 3.14 tornando-se Float -> Float -> Float e Float , respectivamente.

Deve-se notar que as representações visuais sozinhas não tentam mostrar como as classes de tipos estão relacionadas ou quais tipos são membros de quais classes de tipos. Seria claramente desejável que o sistema destacasse quais slots são alvos válidos para qualquer bloco selecionado pelo usuário.

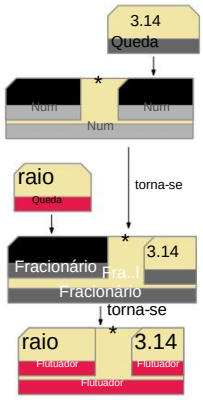

Fig. 6. Comportamento dos blocos com as classes de tipo Num e Fractional.

E. Listas e tuplas

Listas são o tipo de coleção fundamental em linguagens funcionais. Listas em Haskell são coleções homogêneas; uma lista de elementos do tipo a tem o tipo denotado por [a]. As listas podem ser escritas com elementos dados entre colchetes [· · ·], e podem ser construídas usando a lista vazia [] (do tipo [a]) e o operador precedente (cons) ':' do tipo a -> [a] -> [a].
As listas são um tipo parametrizado e isso se reflete na forma de sua representação visual. No entanto, em vez de um nome de cor e tipo, uma área branca é usada, com o tipo de elemento embutido na região superior direita.

A Fig. 7 mostra uma construção de blocos de listas usando a notação [· · ·] e o operador ':'. Observe nas formas dos blocos que ambos podem ser usados dentro dos padrões (eles são considerados como construtores de dados) e que o bloco [· · ·] inclui controles para adicionar e remover slots para elementos.

Tuplas em Haskell são coleções heterogêneas, normalmente contendo apenas 2 ou 3 elementos; um exemplo de valor de tupla é (True, 'a') e tem o tipo (Bool, Char). Os tipos de tupla são representados conforme ilustrado na Fig. 8 com uma base branca estreita e com região(ões) branca(s) curva(s) separando o

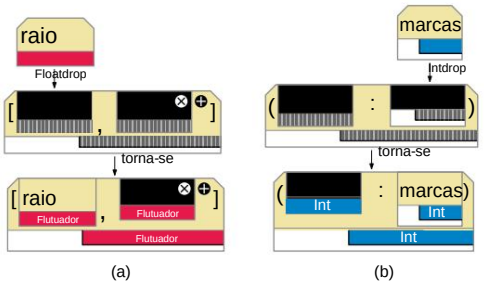

Fig. 7. (a) Uma lista do tipo [Float] construída usando um bloco [· · ·]; (b) uma lista do tipo [Int] construída usando um bloco ':'.

tipos de componentes. Observe o controle para adicionar slots extras (não há controle para remover o segundo slot, pois Haskell não permite tuplas de um elemento).

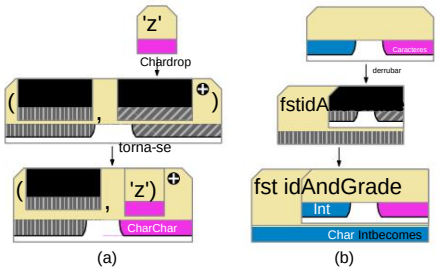

Fig. 8. (a) Um bloco Char é colocado em um slot vazio de um bloco de tupla; (b) um bloco variável do tipo (Int, Char) é inserido no slot de argumento da função de projeção fst :: (a, b) -> a.

As representações visuais para listas (e outros tipos parametrizados) e tuplas claramente permitem o aninhamento de tipos. A Fig. 9 ilustra aninhamentos de listas e de listas com tuplas. O aninhamento pode potencialmente ter qualquer profundidade, mas, é claro, as representações podem se tornar difíceis de interpretar, especialmente para blocos de largura limitada. Esses problemas podem ser atenuados, por exemplo, adicionando espaço em branco extra ao redor do texto de um bloco (quebrando as convenções de formatação) e impondo comprimentos mínimos de nomes de identificadores. Para a maioria das construções de blocos razoáveis, no entanto, as representações devem ser grandes o suficiente para serem claras.

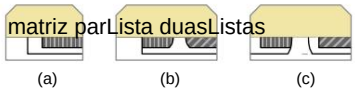

Fig. 9. Listas aninhadas/blocos de variáveis de tupla do tipo (a) [[a]], (b) [(a, b)] e (c) ([a],[b]).

F. Tipos de função Os

tipos de função são representados com uma base branca e uma seta separando os tipos de argumento e resultado. Ilustramos essa representação na Fig. 10 usando os blocos para o operador de composição de função '.' (do tipo (b -> c) -> (a -> b) -> a -> c) e três funções de ordem superior comumente usadas: map :: (a -> b) -> [a] -> [b], filtro :: (a -> Bool) -> [a] -> [a] e foldr :: Dobrável t => (a -> b -> b) -> b -> ta -> b do Prelúdio de Haskell.

O primeiro slot de argumento do bloco foldr mostra o aninhamento de duas funções, ilustrando claramente que o tipo a -> b -> b deve ser interpretado como a -> (b -> b). O terceiro argumento inclui a classe de tipo Foldable da qual as listas são instâncias; assim, por exemplo, o bloco variável de marcas da Fig. 7(b) poderia ser descartado aqui (e as formas do indicador e as representações do tipo de bloco sugerem isso).

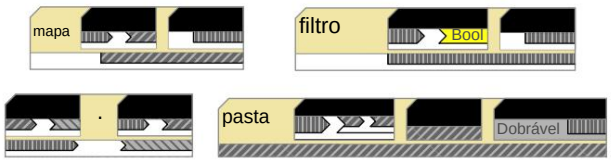

Fig. 10. Blocos para o operador de composição de função '.' e as funções de ordem superior map, filter e foldr.

G. Aplicação de função parcial

A aplicação parcial da função é obtida por meio da adição de controles que permitem que os slots de argumento das funções sejam removidos. Na Fig. 11, o segundo slot do bloco de mapa é removido, resultando em um bloco de um slot (com controles para adicionar novamente o segundo slot e remover o restante (primeiro) slot).
Um operador como '*' também pode ser aplicado parcialmente removendo um ou ambos os seus argumentos para fornecer uma seção de operador. Na Fig. 11, o slot de argumento esquerdo é removido e o slot restante é preenchido com o bloco variável de fator para fornecer a seção do operador (* fator) do tipo Int -> Int.
Soltar este bloco no slot vazio do mapa fornece um bloco completo para o mapa (fator *) (do tipo [Int] -> [Int]).
Observe que os controles no bloco '*' desaparecem quando ele é descartado.

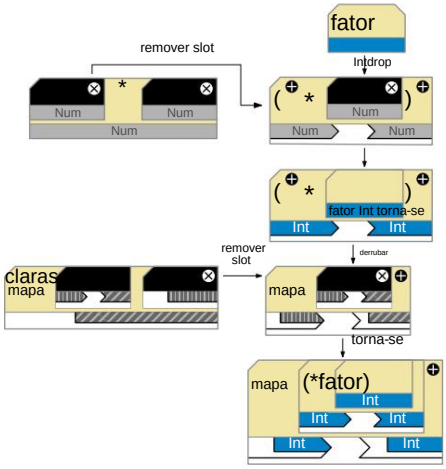

Fig. 11. Construção do mapa de expressão (fator *) através da aplicação parcial de '*' e mapa.

4. CONCLUSÃO E TRABALHO ADICIONAL

Apresentamos um design para blocos que representam expressões em Haskell. As ideias de design destinam-se a formar a base de um ambiente para ajudar os alunos a entender o sistema de tipos de Haskell e a estruturar seu aprendizado de programação funcional. Os próximos passos no projeto são incluir mecanismos para definir funções (principalmente definições locais e guardas), declarações de tipo de função e definições de tipo (algébrico).

REFERÊNCIAS

[1] “Scratch,” scratch.mit.edu, acessado em 11 de julho de 2019.

[2] “Puxa!” snap.berkeley.edu, acessado em 11 de julho de 2019.

[3] M. Poole, “Design de um ambiente baseado em blocos para programação introdutória em Python”, em 2015 Blocks and Beyond Workshop. IEEE, 2015, pp. 31–34.

[4] ——, “Estendendo o design de um ambiente Python baseado em blocos para suportar tipos complexos”, em 2017 Blocks and Beyond Workshop. IEEE, 2017, pp. 1–7.

[5] M. Homer e J. Noble, “Combining tiled and textual views of code,” em Software Visualization (VISSOFT), 2014 Second IEEE Working Conference on Software Visualization. IEEE, 2014, pp. 1–10.

[6] J. Hughes, “Por que a programação funcional é importante,” The Computer Journal, vol. 32, nº. 2, pp. 98–107, 1989.

[7] Z. Hu, J. Hughes e M. Wang, “Como a programação funcional importava,” National Science Review, vol. 2, não. 3, pp. 349–370, 2015.

[8] B. Heeren, D. Leijen e A. van IJzendoorn, "Helium, for learning Haskell", em Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell. ACM, 2003, pp. 62–71.

[9] M. Vasek, “Representando tipos expressivos em linguagens de programação de blocos,” Wellesley College, tese de honra, 2012.

[10] “Bootstrap block editor,” bootstrap-block-editor.appspot.com, acessado em 11 de julho de 2019.

[11] E. Schanzer, S. Krishnamurthi e K. Fisler, “Blocks versus text: On going Lessons from Bootstrap,” em 2015 Blocks and Beyond Workshop. IEEE, 2015, pp. 125–126.

[12] B. Harvey e J. Monig, “Lambda em linguagens de blocos: lições aprendidas,” em 2015 Blocks and Beyond Workshop. IEEE, 2015, pp. 35–38.

[13] S. Kim e F. Turbak, "Adaptando operadores de lista de ordem superior para programação de blocos", em 2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). IEEE, 2015, pp. 213–217.