Seminário sobre F*

DANIELLA MARTINS VASCONCELLOS and LAÍS PISETTA VAN VOSSEN, Universidade do Estado de Santa Catarina, Brasil

O F* (ou FStar ou F Estrela), é uma linguagem de programação versátil que combina tipos dependentes, verificação formal e provas formais. Com sua capacidade de expressar e verificar propriedades sofisticadas dos programas, o F* desempenha um papel fundamental na construção de software seguro e confiável. Através da conexão entre prova e programa, aplicação da teoria de tipos na verificação e o uso de assistentes de provas, o F* oferece uma abordagem abrangente para o desenvolvimento de software confiável, permitindo a expressão de propriedades refinadas dos programas e a verificação formal de sua corretude. Este estudo pretende analisar os básicos da fundamentação da linguagem.

Additional Key Words and Phrases: F*, assistente de provas, programação funcional

ACM Reference Format:
Daniella Martins Vasconcellos and Laís Pisetta Van Vossen. 2023. Seminário sobre F*. 1, 1 (August 2023), 7 pages.

1 INTRODUÇÃO

O F*, também conhecido como FStar (F Estrela) é uma linguagem de programação com tipos dependentes que desempenha vários papéis importantes. Ele funciona como uma linguagem de programação de propósito geral, incentivando a programação funcional de ordem superior com efeitos. Além disso, o F* atua como um compilador que traduz programas escritos em F* para OCaml, F#, C ou Wasm, permitindo sua execução.

Outra função do F* é servir como um assistente de provas, onde é possível afirmar e comprovar propriedades dos programas. Ele também atua como um mecanismo de verificação de programas, utilizando solucionadores SMT para automatizar parcialmente as provas. Além disso, o F* funciona como um sistema de metaprogramação, permitindo a construção programática de programas F* e a automação de procedimentos de prova.

Para alcançar esses objetivos, o design do F* é baseado em alguns elementos-chave. A linguagem central do F* é composta por funções totais com tipos dependentes completos, incluindo recursos como conversão de tipos extensional, tipos indutivos indexados, correspondência de padrões, funções recursivas com verificação semântica de terminação, tipos refinados dependentes, subtipagem e polimorfismo sobre uma hierarquia preditiva de universos. O F* também incorpora um sistema de efeitos indexados definidos pelo usuário e isso inclui a capacidade de lidar com recursão geral, divergência e a definição de efeitos pelo usuário, como estado, exceções, concorrência, efeitos algébricos, entre outros.

Outro aspecto importante do F* é a codificação embutida de um fragmento clássico da lógica do F* na lógica de primeira ordem de um solucionador Satisfiability Modulo Theories (SMT). Isso permite a utilização do solucionador para auxiliar na verificação de programas. Por fim, o F* também oferece a reflexão da sintaxe e do estado de prova do próprio F*, permitindo que programas escritos em Meta-F* manipulem a sintaxe do F* e as metas de prova. Isso possibilita aos usuários construir provas interativamente utilizando táticas.

Authors' address: Daniella Martins Vasconcellos, daniellavasconc@gmail.com; Laís Pisetta Van Vossen, lais.vossen@gmail.com, Universidade do Estado de Santa Catarina, Joinville, Santa Catarina, Brasil.

XXXX-XXXX/2023/8-ART $15.00
https://doi.org/

Então, este artigo se divide em 6 seções. A seção 2 apresenta a história da linguagem e origem de seu nome, além de discutir sobre a maturidade da comunidade que a administra. Na seção 3 são apresentados os elementos principais da sintaxe da linguagem, desde como tipos primitivos são representados até exemplos de seu uso com operações. Na seção 4 são apresentados o conceito de teoria de tipos aplicada à linguagem. Na seção **??**

2 HISTÓRIA DO F* E SUA COMUNIDADE

O F* é um projeto de código aberto no GitHub em desenvolvimento por pesquisadores de várias instituições, incluindo a Microsoft Research, MSR-Inria, Inria, Rosario e Carnegie-Mellon.

O nome *F* no F* é uma homenagem ao *System F* [1] , que era o cálculo base de uma versão anterior do F*. A parte "F" do nome também é derivada de várias linguagens anteriores nas quais muitos autores do F* trabalharam, incluindo Fable, F7, F9, F5, FX e até mesmo F#.

Já o asterisco foi concebido como um tipo de operador de ponto fixo, e o F* foi projetado para ser uma espécie de ponto fixo de todas essas linguagens. A primeira versão do F*[2] também possuía tipos afins, e parte da intenção na época era usar esses tipos afins para codificar a lógica de separação, então o "*" também tinha a intenção de evocar a lógica de separação "*". No entanto, as primeiras versões afins do F* nunca realmente incluíram a lógica de separação. Levou quase uma década para ter a lógica de separação incorporada no F*, embora sem depender de tipos afins.

Diferente de ferramentas como o Isabelle/HOL e o Coq que são ferramentas maduras que estão sendo desenvolvidas e mantidas por várias décadas, possuindo comunidades acadêmicas sólidas e muitas fontes de documentação, o F* é menos maduro, seu design tem sido objeto de diversos artigos de pesquisa, tornando-o um tanto experimental. A comunidade do F* também é menor, sua documentação é mais escassa e os usuários do F* geralmente estão relativamente próximos à equipe de desenvolvimento da linguagem. No entanto, o F* tem crescido, com históricos de adoção industrial e também com uma comunidade ativa no GitHub [2]. A principal fonte de documentação sobre a linguagem se encontra em seu site oficial [3], onde os usuários interessados podem encontrar links para baixar e instalar a linguagem, bem como um livro online em desenvolvimento pela comunidade que apresenta exercícios e exemplos para aqueles que querem aprender mais sobre o F* [1], além de cursos e artigos que utilizaram a linguagem em alguma aplicação.

3 SINTAXE DO F*

Para entender o funcionamento e uso da linguagem F*, é necessário conhecer sua sintaxe básica, compreender como seus tipos primitivos se apresentam, como são feitas funções na linguagem, bem como empregar termos lambdas ou recursões. Para isso, as subseções 3.1 e 3.2 apresentam como programar com o F* em um editor de texto, bem como a estrutura sintática necessária para tal, com base nos tutoriais e capítulos do livro online oficial do F*.

3.1 Editores de Texto

O programas em F* pode ser feitos em qualquer editor de texto, no entanto, a maioria dos usuários avançados do F* utiliza o Emacs[4] em conjunto com o fstar-mode.el[5], que oferece várias utilidades para edição e verificação interativa de arquivos F*, e permite a programação facilitada da linguagem dentro do editor de texto Emacs. O fstar-mode.el depende de um protocolo de interação genérico, porém personalizado, implementado pelo

[1] https://en.wikipedia.org/wiki/System_F
[2] https://github.com/FStarLang/FStar
[3] https://www.fstar-lang.org
[4] https://www.gnu.org/software/emacs/
[5] https://github.com/FStarLang/fstar-mode.el

compilador F*. O F* também oferece uma implementação básica do Language Server Protocol, que pode ser a base para integração com outros editores.

3.2 Estrutura Sintática Básica

Um programa escrito em F* sempre possui módulos, que contêm as definições de funções-base da linguagem. Estes possuem a extensão *.fst* e o nome do arquivo deve ser igual ao nome do módulo a ser importado no programa, então se o módulo importado possui o nome A, o arquivo deverá ser A.fst.

Outra funcionalidade muito utilizada na programação são os comentários, com eles é possível fazer anotações no código para facilitar o entendimento daqueles que o lerem. No F* há duas formas de fazer um comentário, ele pode ser de múltiplas linhas ou de uma única linha, como pode ser visto na Figura 1.

```
(*
Este é um comentário de
múltiplas linhas
*)

// Este é um comentário de uma única linha
```

Fig. 1. Representação de comentários em F*.

As funções no F* são definidas usando a palavra-chave "let" ou "let rec", seguida pelo nome da função, seus argumentos e seu corpo. Além disso, a linguagem possui alguns tipos primitivos definidos nela no módulo *Prims*. Estes tipos são:

- False: este tipo não possui elementos, é o tipo utilizado para proposições que não são prováveis;
- Unit: representação de elementos únicos, definidos como $() : unit$;
- Booleans: o tipo *bool* possui dois possíveis valores, o $true$ e $false$, que representam proposições verdadeiras e falsas respectivamente. Ainda, tipos booleanos possuem os seguintes operadores, em ordem de procedência:
  - not: operador de negação, inverte o valor da proposição, ex.: not true = false;
  - &&: operador de conjunção, ex.: true && false = false;
  - ||: operador de disjunção, ex.: true || false = true.
- Condicionais: elementos que checam valores e condições entre as preposições, são o *if* (se), *then* (então) e *else* (senão), possuem a seguinte sintaxe, conforme apresentado na Figura 2:

```

let condicionais b =
    // se b for verdadeiro, então retorna 1, senão retorna 0
    if b then 1 else 0
```

Fig. 2. Representação de condicionais em F*.

  - Ainda, é possível combinar condicionais com os operadores de booleanos, para verificar valores de combinações de proposições, como exemplificado pela Figura 3
- Inteiros: o tipo *int* representa os elementos do conjunto dos inteiros (0, 1, 2, 3, ...), e possui diversos operadores matemáticos usados para fazer operações sobre inteiros, de forma análoga a contas matemáticas. Estes operadores são apresentados abaixo:

```
val condicionais_operadores (b1:bool) (b2:bool) (b3:bool) : nat
let condicionais_operadores b1 b2 b3=
    (* se b1 e b2 forem verdadeiros, ou b3 for verdadeiro,
    então retorna 18, senão retorna 20 *)
    if b1 && b2 || b3
    then 18
    else 20
```

Fig. 3. Representação de condicionais e operadores combinados em F*.

– -: pode ser utilizado para tornar um valor negativo ou fazer a subtração entre dois valores.
– +: utilizado para a adição entre dois valores;
– /: utilizado para realizar a divisão entre dois inteiros;
– %: operador da função que retorna o resto da divisão
– *op_Multiply*: operação de multiplicação entre inteiros;
– <, <= verifica se o elemento da esquerda é menor que o da direita, no caso do <= verifica se é menor ou igual;
– >, >=: verifica se o elemento da esquerda é maior que o da direita, no caso do >= verifica se é maior ou igual;
– Alguns exemplos de uso desses operadores pode ser visto na Figura 4

```
let maior_que_dobro (x:int) (y:int): bool =
    // verifica se o valor de x é maior que y + y
    x > y + y
let checa_par (x:int) (y:int): bool =
    (* verifica se o resto da divisão
    de x por 2 é menor que 1, ou seja, se x é par *)
    x % 2 < 1
```

Fig. 4. Representação de operações sobre inteiros em F*.

–

A linguagem ainda possui tipos de refinamento booleano, sendo eles uma forma de especificar propriedades mais refinadas sobre os valores em programas. Esses tipos permitem adicionar condições lógicas, expressas por valores *booleanos*, como parte das especificações de tipos.

O módulo *Prims* define os números naturais da seguinte forma:

$$letnat = x : intx >= 0$$

Esta é uma instância de um tipo de refinamento booleano, cuja forma geral é **x:t { e }**, onde **t** é um tipo e **e** é um termo de tipo booleano que pode se referir à variável limitada **x**, de tipo **t**. O termo e refina o tipo **t**, no sentido de que o conjunto **S** denotado por t é restrito aos elementos **x** de **S** para os quais e avalia como verdadeiro.

Isso significa que seria possível descrever o tipo *nat* como o conjunto de termos que avaliam para um elemento do conjunto **0, 1, 2, 3, ...**. No entanto, é possível extrapolar essa definição pode definir refinamentos arbitrários de qualquer escolha, como seguem os exemplos da figura 5.

4 VERIFICAÇÃO DE PROGRAMAS

A principal vantagem dos tipos dependentes é a capacidade de especificar e verificar propriedades mais precisas sobre os programas. Em F*, é possível definir tipos que dependem de valores, o que permite que propriedades

```
let empty = x:int { false } // conjunto vazio
let zero = x:int{ x = 0 } // tipo que contem um elemento 0
let pos = x:int { x > 0 } // os números positivos
let neg = x:int { x < 0 } // os números negativos
let even = x:int { x % 2 = 0 } // os números pares
let odd = x:int { x % 2 = 1 } // os números ímpares
```

Fig. 5. Representação de conjuntos possíveis em F*

sejam expressas de forma mais detalhada e verificadas pelo compilador. Por exemplo, é possível especificar que uma lista está ordenada de forma crescente ou que um número inteiro é par. Essas propriedades podem ser verificadas estaticamente, ou seja, antes mesmo da execução do programa. Além disso, a verificação formal em F* permite o raciocínio sobre a corretude e segurança dos programas. É possível especificar pré e pós-condições, invariáveis e lemas, e usar o sistema de tipos para verificar se essas propriedades são satisfeitas. Isso proporciona uma garantia mais forte de que o programa está correto e protegido contra falhas de segurança.

Para ilustrar a verificação formal em F*, consideremos um exemplo de verificação de um programa simples. Suponha que desejamos verificar se uma função de ordenação de uma lista retorna sempre uma lista ordenada. Podemos especificar essa propriedade utilizando tipos dependentes e escrever provas que demonstrem essa propriedade. O compilador de F* pode então verificar se as provas são válidas, fornecendo uma garantia formal da corretude do programa.

5 PROVAS E TEORIA DE TIPOS

Em F*, a conexão entre prova e programa é estabelecida por meio da utilização de lemas e provas formais. Um lema em F* é uma função que sempre retorna o valor **():unit**, mas o tipo do lema carrega informações úteis sobre fatos que são prováveis. Por exemplo, podemos definir um lema para provar que a função fatorial retorna um número positivo para argumentos maiores do que zero. O tipo do lema expressa essa propriedade, garantindo que, quando o lema for invocado, a conclusão factorial x > 0 seja válida. Dessa forma, a prova formal de um lema estabelece uma conexão entre a propriedade desejada e o programa correspondente.

A teoria de tipos desempenha um papel crucial na verificação de programas em F*, pois permite a especificação precisa de tipos dependentes e refinamentos booleanos, que restringem o comportamento dos programas e garantem propriedades estáticas. Em F*, é possível definir tipos refinados para expressar propriedades específicas dos programas, como listas ordenadas ou números pares, o que ajuda a evitar erros e fornece uma verificação estática das propriedades dos programas. Além disso, a teoria de tipos estabelece as regras para inferência de tipos, permitindo que o sistema de tipos do F* verifique automaticamente a correção dos programas em relação às propriedades especificadas.

Em F*, é comum o uso de assistentes de provas para auxiliar na verificação formal de programas. Os assistentes de provas permitem que os desenvolvedores construam provas formais para demonstrar a corretude e a segurança dos programas. No contexto do F*, os assistentes de provas podem ser utilizados para estabelecer a validade de lemas, realizar provas por indução e aplicar estratégias de raciocínio lógico. Essas ferramentas auxiliam os desenvolvedores a construir argumentos formais robustos, verificando a correção dos programas e garantindo que eles se comportem de acordo com as propriedades especificadas.

6 EXEMPLO PRÁTICO: QUICKSORT

Para estudo, utilizaremos as definições de provas de lemas para demonstrar a correção do *Quicksort*, um algoritmo clássico de ordenação.

Começaremos com listas de números inteiros e descreveremos algumas propriedades que desejamos que um algoritmo de ordenação cumpra, o mostrado na imagem . Iniciaremos com a função “sorted”, que determina se uma lista de números inteiros está ordenada em ordem crescente, e a função “mem”, que verifica se um determinado elemento está presente em uma lista. É possível observar que “mem” utiliza um “eqtype”, que é o tipo de tipos que suportam igualdade decidível. A função “sorted” verifica se a lista está vazia, contém apenas um elemento ou se cada elemento é menor ou igual ao próximo na lista. Enquanto a função “mem” verifica se um elemento está presente na lista percorrendo-a recursivamente. Dado um algoritmo de ordenação chamado “sort”, gostaríamos de provar a seguinte propriedade: para todas as listas de entrada “l”, a lista resultante de “sort l” estará ordenada e conterá todos os elementos presentes em “l”.

Sendo assim, a implementação do Quicksort se dá da seguinte maneira: Sempre escolhe o primeiro elemento da lista como pivô, divide o restante da lista em elementos maiores ou iguais ao pivô e os demais, ordena recursivamente as partições e insere o pivô no meio antes de retornar. A implementação do código se mostra na imagem 6.

```
let rec sorted (l:list int)
  : bool
  = match l with
    | [] -> true
    | [x] -> true
    | x :: y :: xs -> x <= y && sorted (y :: xs)

let rec mem (#a:eqtype) (i:a) (l:list a)
  : bool
  = match l with
    | [] -> false
    | hd :: tl -> hd = i || mem i tl

forall l. sorted (sort l) /\ (forall i. mem i l <==> mem i (sort l))

let rec sort (l:list int)
  : Tot (list int) (decreases (length l))
  = match l with
    | [] -> []
    | pivot :: tl ->
      let hi, lo  = partition ((<=) pivot) tl in
      append (sort lo) (pivot :: sort hi)
```

Fig. 6. Código de demonstração: Quicksort

7 INTEGRAÇÃO COM OUTRAS LINGUAGENS

Embora o F* seja uma linguagem poderosa para especificação e verificação formal, nem sempre é prático escrever programas completos apenas em F*. Torna-se útil a integração com outras linguagens, tornando a linguagem flexível para especificação e verificação formal, permitindo que o usuário aproveite as vantagens da verificação e da especificação precisa enquanto mantém a flexibilidade de usar outras linguagens onde necessário. Essa interoperabilidade permite usar o F* para especificar componentes críticos de um sistema e enquanto aproveita as outras linguagens para implementar partes não críticas ou para obter melhor desempenho.

Algumas das linguagens que possuem boa integração com o F* são:

- **C:** O F* possui um back-end de geração de código C, o que significa que se pode escrever especificações em F* e gerar código C correspondente. Isso é especialmente útil quando há necessidade de alto desempenho

ou de interação com sistemas existentes escritos em C. O F* cuida da verificação formal das especificações, enquanto o código C gerado é responsável pela implementação real.

- **OCaml:** O F* foi desenvolvido como uma extensão do OCaml e compartilha muitas semânticas e recursos com a linguagem. Isso permite que se incorpore facilmente código OCaml em seus programas F* e vice-versa. É possível escrever especificações em F* e usar bibliotecas OCaml existentes para implementar partes não críticas ou explorar a vasta base de código disponível em OCaml.
- **F#:** O F# é um dialeto do OCaml que roda na plataforma .NET. Assim como a integração com OCaml, é possível usar código F# em conjunto com o F*. Essa interoperabilidade permite que se tire proveito da biblioteca e da infraestrutura .NET existentes ao escrever especificações em F*.

8 CONCLUSÕES

Neste artigo, foi explorado os principais aspectos do F*, uma linguagem de programação poderosa e versátil, que oferece uma abordagem inovadora para a programação funcional e a verificação de programas. O F* desempenha vários papéis importantes, atuando como uma linguagem de propósito geral, um compilador para diferentes linguagens de destino e um assistente de provas.

A linguagem central do F* é composta por funções totais com tipos dependentes completos, incluindo recursos como tipos indutivos indexados, correspondência de padrões e tipos refinados dependentes, oferecendo um sistema de efeitos indexados, o que permite lidar com efeitos como estado, exceções e concorrência. O F* é um importante assistente de provas e um mecanismo de verificação de programas, já que sua integração com solucionadores SMT permite automatizar parcialmente as provas e verificar propriedades dos programas de forma confiável. Relevante relembrar que o F* oferece recursos de metaprogramação, possibilitando a construção programática de programas e a automação de procedimentos de prova.

Embora o F* ainda seja considerado uma linguagem experimental, seu potencial é promissor. Através de sua combinação única de recursos, o F* abre caminho para o desenvolvimento de software mais seguro e confiável, além de possibilitar avanços na área de provas formais. Com uma comunidade em constante crescimento e recursos de documentação em desenvolvimento, o F* continua a atrair a atenção de pesquisadores e desenvolvedores interessados em explorar os benefícios da programação com tipos dependentes e verificação de programas.

REFERÊNCIAS

[1] 2020. *Proof-oriented Programming in F**. Microsoft Research.

[2] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F*. *SIGPLAN Not.* 51, 1 (jan 2016), 256–270. https://doi.org/10.1145/2914770.2837655

Received 15 de agosto de 2023