Universidade da Beira Interior 2020–2021

Trabalho de Compilação

Pico Rust

O objectivo deste trabelho é implementar um compilador para um fragmento da linguagem de programação Rust, designado de Pico Rust no resto do documento, que produz código x86-64.

Trata-se de um fragmento contendo inteiros, apontadores vetores e estuturas. Trata-se de um fragmento 100% compatível com Rust, no sentido que todo o programa Pico Rust é também um programa Rust correto. Tal permitirá usar um compilador Rust como compilador de referência.

O presente documento descreve a sintaxe de Pico Rust.

1 Sintaxe

Neste documento utilisaremos as notações seguintes nas gramáticas :

$\langle regra \rangle^\star$	repetição da regra $\langle regra \rangle$ um número qualquer de vezes (incluído nenhuma vez)
$\langle regra \rangle_t^\star$	repetição da regra $\langle regra \rangle$ um número qualquer de vezes (incluído nenhuma vez), as ocorrências sendo separadas pelo terminal t
$\langle regra \rangle^+$	repetição da regra $\langle regra \rangle$ pelo menos uma vez
$\langle regra \rangle_t^+$	repetição da regra $\langle regra \rangle$ pelo menos uma vez, as ocorrências sendo separadas terminal t
$\langle regra \rangle?$	utilização opcional da regra $\langle regra \rangle$ (*i.e.* 0 ou 1 vez)
($\langle regra \rangle$)	colocar entre parênteses

Tenha atenção em não confundir " * " e " $^+$ " com " `*` " e " `+` " que são símbolos da linguagem Rust.

Tenha também atenção em não confundir as parênteses (gramaticais) com os terminais `(` et `)` da linguagem.

1.1 Convenções léxicas

Espaços, tabulações e *carrier-return* são os *caracteres brancos*. Os comentários podem tomar duas formas :

- começam com `/*`, extendam-se até `*/`, possivelmente aninhados ;
- começam com `//` e extendam-se até ao fim de linha.

Os identificadores seguem a expressão regular $\langle ident \rangle$ seguintes :

$$
\begin{array}{lll}
\langle algarismo \rangle & ::= & \texttt{0–9} \\
\langle alpha \rangle & ::= & \texttt{a–z} \mid \texttt{A–Z} \\
\langle ident \rangle & ::= & \langle alpha \rangle \ (\langle alpha \rangle \mid \langle algarismo \rangle \mid \texttt{\_})^\star
\end{array}
$$

Os identificadores seguintes são palavras chaves :

```
else false fn if let mut return struct true while
```

Por fim, as constantes literais (os inteiros ou as cadeias de caracteres) seguem as expressões regulares ⟨*inteiro*⟩ et ⟨*cadeia*⟩ seguintes :

⟨*inteiro*⟩ ::= ⟨*algarismo*⟩⋆
⟨*caractere*⟩ ::= qualquer caractere diferente de \ e de "
| \\ | \" | \n
⟨*cadeia*⟩ ::= " ⟨*caractere*⟩⋆ "

1.2 Sintaxe

A gramática dos ficheiros fontes é dada na figura 1. O ponto de entrada é o não-terminal ⟨*ficheiro*⟩. As associatividades e precedências dos diversos operadores são dadas pela tabela abaixo, da mais fraca à mais forte precedência.

operador	associatividade	precedência
`=`	direita	fraca
`||`	esquerda	
`&&`	esquerda	
`==` `!=` `<` `<=` `>` `>=`	—	↓
`+` `-`	esquerda	
`*` `/` `%`	esquerda	
`!` `*` (unária) `-` (unária) `&` `&mut`	—	
`[]`	—	
`.`	—	forte

2 Tipagem estática

Após a análise estática se concluir com sucesso, verifica-se a consistência do ficheiro fonte.

2.1 Tipos e ambientes

No que se segue, as expressões de tipos são da forma seguinte :

$$\tau ::= \texttt{()} \mid \texttt{i32} \mid \texttt{bool} \mid S \mid \texttt{Vec<}\tau\texttt{>} \mid \texttt{\&}m\,\tau$$

onde S designa um identificador de estrutura e m um indicador de mutabilidade, isto é `mut` ou nada. Trata-se de uma notação para a sintaxe *abstrata* das expressões de tipo. Um tipo da forma $\texttt{\&}m\,\tau$ é designado por *tipo empréstimo* (*borrow* em inglês).

Introduzimos nos tipos a relação $\tau_1 \leq \tau_2$ que significa " qualquer valor de tipo τ_1 pode ser utilizada onde se espera um valor de tipo τ_2 ". Esta é definida pelas regras seguintes :

$$\frac{}{\tau \leq \tau} \qquad \frac{}{\texttt{\&mut}\,\tau \leq \texttt{\&}\,\tau}$$

Um ambiente de tipagem Γ é uma sequência de declarações de variáveis da forma $m\,x : \tau$, de declarações de estruturas da forma $\texttt{struct}\ S\ \texttt{\{}x_1 : \tau_1, \ldots, x_n : \tau_n\texttt{\}}$ e de declarações de perfis de função da forma $f(\tau_1, \ldots, \tau_n) \to \tau$. Utilizar-se-á a notação $\texttt{struct}\ S\ \texttt{\{}x : \tau\texttt{\}}$ para indicar que a estrutura S contém um campo x de tipo τ.

Diz-se que um tipo τ está *bem formado* num ambiente de tipagem Γ, notação $\Gamma \vdash \tau$ `bf`, se todos os identificadores de estruturas que apareçam em τ correspondem a estruturas declarados em Γ.

$$
\begin{array}{lcl}
\langle \textit{ficheiro} \rangle & ::= & \langle \textit{decl} \rangle^{\star}\ \texttt{EOF} \\
\langle \textit{decl} \rangle & ::= & \langle \textit{decl\_fun} \rangle \mid \langle \textit{decl\_struct} \rangle \\
\langle \textit{decl\_struct} \rangle & ::= & \texttt{struct}\ \langle \textit{ident} \rangle\ \texttt{\{}\ (\langle \textit{ident} \rangle\ \texttt{:}\ \langle \textit{type} \rangle)^{\star}_{\texttt{,}}\ \texttt{\}} \\
\langle \textit{decl\_fun} \rangle & ::= & \texttt{fn}\ \langle \textit{ident} \rangle\ \texttt{(}\ \langle \textit{argument} \rangle^{\star}_{\texttt{,}}\ \texttt{)}\ (\texttt{->}\ \langle \textit{type} \rangle)?\ \langle \textit{bloco} \rangle \\
\langle \textit{type} \rangle & ::= & \langle \textit{ident} \rangle \mid \langle \textit{ident} \rangle\ \texttt{<}\ \langle \textit{type} \rangle\ \texttt{>} \mid \texttt{\&}\ \langle \textit{type} \rangle \mid \texttt{\&}\ \texttt{mut}\ \langle \textit{type} \rangle \\
\langle \textit{argument} \rangle & ::= & \texttt{mut}?\ \langle \textit{ident} \rangle\ \texttt{:}\ \langle \textit{type} \rangle \\
\langle \textit{bloco} \rangle & ::= & \texttt{\{}\ \langle \textit{instr} \rangle^{\star}\ \langle \textit{expr} \rangle?\ \texttt{\}} \\
\langle \textit{instr} \rangle & ::= & \texttt{;} \\
 & \mid & \langle \textit{expr} \rangle\ \texttt{;} \\
 & \mid & \texttt{let}\ \texttt{mut}?\ \langle \textit{ident} \rangle\ \texttt{=}\ \langle \textit{expr} \rangle\ \texttt{;} \\
 & \mid & \texttt{let}\ \texttt{mut}?\ \langle \textit{ident} \rangle\ \texttt{=}\ \langle \textit{ident} \rangle\ \texttt{\{}\ (\langle \textit{ident} \rangle\ \texttt{:}\ \langle \textit{expr} \rangle)^{\star}_{\texttt{,}}\ \texttt{\}}\ \texttt{;} \\
 & \mid & \texttt{while}\ \langle \textit{expr} \rangle\ \langle \textit{bloco} \rangle \\
 & \mid & \texttt{return}\ \langle \textit{expr} \rangle?\ \texttt{;} \\
 & \mid & \langle \textit{if} \rangle \\
\langle \textit{if} \rangle & ::= & \texttt{if}\ \langle \textit{expr} \rangle\ \langle \textit{bloco} \rangle\ (\texttt{else}\ (\langle \textit{bloco} \rangle \mid \langle \textit{if} \rangle))? \\
\langle \textit{expr} \rangle & ::= & \langle \textit{entier} \rangle \mid \texttt{true} \mid \texttt{false} \\
 & \mid & \langle \textit{ident} \rangle \\
 & \mid & \langle \textit{expr} \rangle\ \langle \textit{binário} \rangle\ \langle \textit{expr} \rangle \\
 & \mid & \langle \textit{unário} \rangle\ \langle \textit{expr} \rangle \\
 & \mid & \langle \textit{expr} \rangle\ \texttt{.}\ \langle \textit{ident} \rangle \\
 & \mid & \langle \textit{expr} \rangle\ \texttt{.}\ \texttt{len}\ \texttt{(}\ \texttt{)} \\
 & \mid & \langle \textit{expr} \rangle\ \texttt{[}\ \langle \textit{expr} \rangle\ \texttt{]} \\
 & \mid & \langle \textit{ident} \rangle\ \texttt{(}\ \langle \textit{expr} \rangle^{\star}_{\texttt{,}}\ \texttt{)} \\
 & \mid & \texttt{vec}\ \texttt{!}\ \texttt{[}\ \langle \textit{expr} \rangle^{\star}_{\texttt{,}}\ \texttt{]} \\
 & \mid & \texttt{print}\ \texttt{!}\ \texttt{(}\ \langle \textit{cadeia} \rangle\ \texttt{)} \\
 & \mid & \langle \textit{bloco} \rangle \\
 & \mid & \texttt{(}\ \langle \textit{expr} \rangle\ \texttt{)} \\
\langle \textit{binário} \rangle & ::= & \texttt{==} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \\
 & \mid & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{\&\&} \mid \texttt{||} \mid \texttt{=} \\
\langle \textit{unário} \rangle & ::= & \texttt{-} \mid \texttt{!} \mid \texttt{*} \mid \texttt{\&} \mid \texttt{\&}\ \texttt{mut}
\end{array}
$$

Figura 1: Gramática dos ficheiros Pico Rust.

2.2 Regras de tipagem

Denota-se por τ_r o tipo de retorno de uma função que se encontra em fase de verificação. Introduzimos três juízos de tipagem seguintes :

$\Gamma \vdash e : \tau$, para " a expressão e está bem tipada e é de tipo τ " ;

$\Gamma \vdash_l e : \tau$, para " a expressão e é um valor esquerdo bem tipado e tem por tipo τ " ;

$\Gamma \vdash_{\texttt{mut}} e$, para " a expressão e é mutável ".

Estes juízos de tipos são definidos pelas regras de inferência seguintes :

$$\frac{c \text{ constante de tipo } \tau}{\Gamma \vdash c : \tau} \qquad \frac{m\, x : \tau \in \Gamma}{\Gamma \vdash_l x : \tau} \qquad \frac{\Gamma \vdash_l e : \tau}{\Gamma \vdash e : \tau}$$

$$\frac{\Gamma \vdash_l e_1 : \tau_1 \quad \Gamma \vdash_{\texttt{mut}} e_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_2 \leq \tau_1}{\Gamma \vdash e_1 \texttt{ = } e_2 : \texttt{()}}$$

$$\frac{\Gamma \vdash e : \texttt{i32}}{\Gamma \vdash \texttt{-}\ e : \texttt{i32}} \qquad \frac{\Gamma \vdash e : \texttt{bool}}{\Gamma \vdash \texttt{!}\ e : \texttt{bool}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{i32} \quad \Gamma \vdash e_2 : \texttt{i32} \quad op \in \{\texttt{==}, \texttt{!=}, \texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}\}}{\Gamma \vdash e_1\ op\ e_2 : \texttt{bool}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{i32} \quad \Gamma \vdash e_2 : \texttt{i32} \quad op \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}\}}{\Gamma \vdash e_1\ op\ e_2 : \texttt{i32}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \quad \Gamma \vdash e_2 : \texttt{bool} \quad op \in \{\texttt{||}, \texttt{\&\&}\}}{\Gamma \vdash e_1\ op\ e_2 : \texttt{bool}}$$

$$\frac{\Gamma \vdash_l e : \tau}{\Gamma \vdash \texttt{\&}\, e : \texttt{\&}\, \tau} \qquad \frac{\Gamma \vdash_l e : \tau \quad \Gamma \vdash_{\texttt{mut}} e}{\Gamma \vdash \texttt{\&mut}\, e : \texttt{\&mut}\, \tau} \qquad \frac{\Gamma \vdash e : \texttt{\&}m\, \tau}{\Gamma \vdash_l \texttt{*}e : \tau}$$

$$\frac{\forall i.\ \Gamma \vdash e_i : \tau}{\Gamma \vdash \texttt{vec![}e_1, \ldots, e_n\texttt{]} : \texttt{Vec<}\tau\texttt{>}} \qquad \frac{\Gamma \vdash e : \texttt{Vec<}\tau\texttt{>}}{\Gamma \vdash e\texttt{.len()} : \texttt{i32}} \qquad \frac{\Gamma \vdash_l e_1 : \texttt{Vec<}\tau\texttt{>} \quad \Gamma \vdash e_2 : \texttt{i32}}{\Gamma \vdash_l e_1\texttt{[}e_2\texttt{]} : \tau}$$

$$\frac{\texttt{struct}\ S\ \texttt{\{}x_1 : \tau_1, \ldots, x_n : \tau_n\texttt{\}} \in \Gamma \quad \pi \text{ uma permutação} \quad \forall i.\ \Gamma \vdash e_i : \tau_i}{\Gamma \vdash S\texttt{\{}x_{\pi(1)} : e_{\pi(1)}, \ldots, x_{\pi(n)} : e_{\pi(n)}\texttt{\}} : S}$$

$$\frac{\Gamma \vdash_l e : S \quad \texttt{struct}\ S\ \texttt{\{}x : \tau\texttt{\}} \in \Gamma}{\Gamma \vdash_l e\texttt{.}x : \tau}$$

$$\frac{f(\tau'_1, \ldots, \tau'_n) \to \tau \in \Gamma \quad \forall i.\ \Gamma \vdash e_i : \tau_i \quad \tau_i \leq \tau'_i}{\Gamma \vdash f(e_1, \ldots, e_n) : \tau} \qquad \frac{}{\Gamma \vdash \texttt{print!(}s\texttt{)} : \texttt{()}}$$

$$\frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{if}\ e\ e_1\ \texttt{else}\ e_2 : \tau} \qquad \frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash e_1 : \texttt{()}}{\Gamma \vdash \texttt{while}\ e\ e_1 : \texttt{()}}$$

$$\frac{}{\Gamma \vdash \texttt{return} : \texttt{()}} \qquad \frac{\Gamma \vdash e : \tau_r}{\Gamma \vdash \texttt{return}\ e : \texttt{()}}$$

$$\frac{}{\Gamma \vdash \texttt{\{\}} : \texttt{()}} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{\{}\ e\ \texttt{\}} : \tau} \qquad \frac{\Gamma \vdash \texttt{\{}\ b\ \texttt{\}} : \tau}{\Gamma \vdash \texttt{\{}\ \texttt{;}\ b\ \texttt{\}} : \tau}$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma, m\, x : \tau \vdash \texttt{\{}\ b\ \texttt{\}} : \tau'}{\Gamma \vdash \texttt{\{ let}\ m\ x = e\texttt{;}\ b\ \texttt{\}} : \tau'} \qquad \frac{e \neq \texttt{let} \quad \Gamma \vdash e : \tau \quad \Gamma \vdash \texttt{\{}\ b\ \texttt{\}} : \tau'}{\Gamma \vdash \texttt{\{}\ e\texttt{;}\ b\ \texttt{\}} : \tau'}$$

Nas regras finais, b designa o conteúdo dum bloco, isto é uma sequência de expressão separadas por ponto-e-vírgula, eventualmente terminada por uma última expressão que se termina sem ponto-e-vírgula.

Junta-se também três regras de *auto-dereferenciação*, que permitam respectivamente escrever $e\,.\,x$ no lugar de $(*e)\,.\,x$, $e\texttt{[}e_2\texttt{]}$ em vez de $(*e)\texttt{[}e_2\texttt{]}$ e $e\texttt{.len()}$ no lugar de $(*e)\texttt{.len()}$. Estas regras são as seguintes :

$$\frac{\Gamma \vdash e : \&m\,\texttt{Vec<}\tau\texttt{>}}{\Gamma \vdash e\texttt{.len()} : \texttt{i32}} \qquad \frac{\Gamma \vdash e : \&m\,\texttt{Vec<}\tau\texttt{>} \quad \Gamma \vdash e_2 : \texttt{i32}}{\Gamma \vdash_l e\texttt{[}e_2\texttt{]} : \tau}$$

$$\frac{\Gamma \vdash e : \&m\,S \quad \texttt{struct}\ S\ \texttt{\{}x : \tau\texttt{\}} \in \Gamma}{\Gamma \vdash_l e\,.\,x : \tau}$$

Finalmente as regras sobre a mutabilidade são as seguintes :

$$\frac{\texttt{mut}\ x : \tau \in \Gamma}{\Gamma \vdash_{\texttt{mut}} x} \qquad \frac{\Gamma \vdash_{\texttt{mut}} e}{\Gamma \vdash_{\texttt{mut}} e\texttt{[}e_2\texttt{]}} \qquad \frac{\Gamma \vdash_{\texttt{mut}} e}{\Gamma \vdash_{\texttt{mut}} e\,.\,x} \qquad \frac{\Gamma \vdash e : \texttt{\&mut}\,\tau}{\Gamma \vdash_{\texttt{mut}} *e}$$

Nestas ultimas regras, supõe-se que a auto-dereferenciação já foi aplicada, isto é que e foi substituído por $*e$ em $e\,.\,x$ ou em $e\texttt{[}e_2\texttt{]}$ se e tem por tipo $\&m\,\tau$.

2.3 Tipagem dos ficheiros

Relembramos que um ficheiro é uma lista de declarações. Introduzimos o juízo $\Gamma \vdash d \Rightarrow \Gamma'$ que significa " no ambiente Γ, a declaração d é bem formada e produz um ambiente Γ' ". Este juízo é derivável a custa das regras seguintes.

Declaração de uma estrutura. A validade de uma declaração de estrutura consiste essencialmente em validar os tipos dos seus campos :

$$\frac{\forall i.\ \Gamma, \texttt{struct}\ S\ \texttt{\{}x_1 : \tau_1, \ldots, x_n : \tau_n\texttt{\}} \vdash \tau_i\ \mathsf{bf} \quad \tau_i \text{ não contém empréstimos}}{\Gamma \vdash \texttt{struct}\ S\ \texttt{\{}x_1 : \tau_1, \ldots x_n : \tau_n\texttt{\}} \Rightarrow \{\texttt{struct}\ S\ \texttt{\{}x_1 : \tau_1, \ldots x_n : \tau_n\texttt{\}}\} \cup \Gamma}$$

Verificar-se-á também que os tipos τ_i dos campos fazem referência à própria estrutura S somente por um tipo `Vec`. Verificar-se-á finalmente a unicidade dos identificadores $x_1, \ldots, x_n$.

Declarações de funções. Quando um tipo de retorno τ_r de uma função é omitida na sintaxe, trata-se do tipo `()`.

$$\frac{\begin{array}{c}\Gamma \vdash \tau_r\ \mathsf{bf} \quad \tau_r \text{ não contém empréstimos} \quad \forall i.\ \Gamma \vdash \tau_i\ \mathsf{bf} \\ \{f(\tau_1, \ldots, \tau_n) \to \tau_r,\ x_1 : \tau_1, \ldots, x_n : \tau_n\} \cup \Gamma \vdash \texttt{\{}b\texttt{\}} : \tau \\ \tau = \texttt{()} \text{ e a avaliação de } b \text{ conduz sempre a um } \texttt{return}\text{, ou então a } \tau = \tau_r\end{array}}{\Gamma \vdash \texttt{fn}\ f(x_1 : \tau_1, \ldots, x_n : \tau_n) \to \tau_r\ \texttt{\{}b\texttt{\}} \Rightarrow \{f(\tau_1, \ldots, \tau_n) \to \tau_r\} \cup \Gamma}$$

Nota-se que o protótipo de uma função é acrescentado ao ambiente para a tipagem de esta última, para que se possa aceitar as funções recursivas. Verifica-se igualmente a unicidade dos identificadores $x_1, \ldots, x_n$.

Ficheiros. Introduz-se finalmente o juizo $\Gamma \vdash_f d_1 \cdots d_n$ que significa " no ambiente Γ o ficheiro constituídos pela sequência de declarações $d_1, \ldots, d_n$ está bem formado ".

Dar um tipo a um ficheiro consiste assim em dar um tipo sucessivamente a todas as declarações no contexto estendido por cada nova declaração. Donde as regras :

$$\frac{}{\Gamma \vdash_f \emptyset} \qquad \frac{\Gamma \vdash d_1 \Rightarrow \Gamma' \quad \Gamma' \vdash_f d_2 \cdots d_n}{\Gamma \vdash_f d_1\ d_2 \cdots d_n}$$

Verificar-se-á também a unicidade dos identificadores de estruturas e a unicidade dos identificadores de funções, ambas na globalidade do ficheiro.

Ponto de entrada. Por fim, é necessário verificar a presença de uma função `main` com o perfil `fn main()`.

2.4 Tipagem dos recursos

O principal interesse da linguagem Rust, em relação à linguagem C, está nas garantias fortes que a sua tipagem traz à gestão dos recursos, em particular da memória. Por exemplo, a tipagem de Rust garante que nunca se procurará a libertar duas vezes o mesmo bloco de memória alocado na *heap* (double `free`) ou ainda que nunca se seguirá uma referência fantasma (uma referência para um bloco de memória que foi entretanto liberto).

Esta secção descreve as verificações suplementares. Trata-se de verificações *estáticas*, isto é, é efectuada durante a compilação.

Sugerimos que essas sejam feitas durante uma segunda fase de tipagem, após que todas as verificações descritas das secções anteriores sejam realizadas.

Os valores estão separadas em duas famílias, conforme os seus tipos.

Os valores de tipo `()`, `i32`, `bool` ou `&`τ são ditos *duplicáveis* (*copy* em inglês). Os valores de tipo `&mut` τ, estrutura ou `Vec<`τ`>` são ditos *movíveis* (*move* em inglês). Na sequência, chamamos de *recurso* todo o valor desta segunda categoria.

Qualquer recurso tem um único dono, que é uma variável. Quando uma variável proprietária atinge o fim da sua zona de alcance, liberta-se o recurso e este deixa de estar acessível. O porte (ou, alternativamente, o alcance) de uma variável é assim definida : o porte de uma variável introduzida por `let` estende-se até ao fim do bloco onde esta é definida; o porte de um parâmetro formal estende-se até ao fim da função. No fim de um bloco, os recursos são devolvidos (libertos) na ordem inversa da declaração dos seus donos.

O dono de um recurso v pode mudar com o tempo, aquando

- de uma atribuição da forma $l = v$;
- de uma atribuição para uma variável `let` x `=` v`;` ;
- de uma atribuição implícita na construção de um vector `vec![`$\ldots, v, \ldots$`]` ou de uma estrutura S `{`$\ldots,\ c : v, \ldots$`}` ;
- de uma passagem como parâmetro efectivo de uma chamada $f(\ldots, v, \ldots)$.

Consideremos os dois programas seguintes :

```
let v = vec![1,2,3];
let w = v;
let a = w[0];
```

```
let v = vec![1,2,3];
let w = v;
let a = v[0];
```

O programa da esquerda é aceite porque acedemos ao recurso (o vector) pelo seu novo dono `w`. Mas o programa da direita é recusado porque tentamos aceder ao recurso pelo intermédio do seu antigo dono `v`.

Para evitar mudanças intrusivas de proprietários na prática, Rust introduz a noção de *referência*. Uma referência é um apontador para um valor de tipo τ e o seu tipo é `&`$m\,\tau$. O booleano m indica se a referência é imutável (de tipo `&`τ) ou ao contrário mutável (de tipo `&mut` τ). A criação de uma referência para um recurso v de tipo τ, com a construção `&`$m\,v$ designada de "empréstimo", deve seguir várias restrições :

- o porte de uma referência não deve exceder o do proprietário do recurso ;

- podem existir várias referências imutáveis (`&` τ) para o mesmo recurso num dado momento mas só há no máximo uma referência mutável (`&mut` τ).

Nos testes fornecidos, o directório `typing2/` contém os testes específicos a tipagem dos recursos (testes positivos em `typing2/good/` e testes negativos em `typing2/bad/`).

2.5 Exposição adicional.

Como exposto anteriormente, no contexto da tipagem dos recursos para Pico Rust, limitámo-nos à granularidade de uma variável.

Em particular, não podemos expressar em Pico Rust o empréstimo de um dado campo de uma estrutura mas somente o da estrutura como um todo. O princípio da tipagem dos recursos consiste em associar, em todo o ponto de uma programa, um estatuto a cada variável (visível neste referido ponto do programa). Este estatuto tem três valores possíveis :

$$\mathit{statut} ::= \texttt{Vazio} \mid \texttt{Cheio} \mid \texttt{Emprestado}(m)$$

O estatuto `Vazio` significa que o conteúdo da variável foi deslocado e não pode mais ser usado (pode, no entanto, este conteúdo ser substituído por outra coisa). O estatuto `Cheio` significa que a variável é dona do recurso ; podemos assim fazer o que bem nos entende com ela. Finalmente, o estatuto `Emprestado`(m) significa que o conteúdo da variável foi emprestado, de forma mutável se $m =$ `mut`.

Na linguagem Pico Rust, um *prazo de vida* coincide com um bloco, sendo uma chamada considerada como fazendo-se num novo bloco implícito (é o bloco que corresponde ao corpo da função invocada)[1]. Os prazos de vida denotam-se por α, β, etc. Introduzimos uma relação de ordem parcial sobre os prazos de vida, denotada $\alpha \leq \beta$, que significa " o prazo de vida β contém o prazo de vida α ".

Durante a tipagem dos recursos, vamos associar a cada expressão um tipo enriquecido com a informação dos prazos de vida ao nível de cada empréstimo, isto é, um tipo da forma :

$$\tau ::= \texttt{()} \mid \texttt{i32} \mid \texttt{bool} \mid S \mid \texttt{Vec<}\tau\texttt{>} \mid \texttt{&}^{\alpha} m\, \tau$$

Introduzimos o juizo $\alpha \vdash \tau$ bf que significa " o tipo τ é válido para o prazo de vida α " e definimo-lo assim :

$$\frac{\tau \in \{\texttt{()}, \texttt{i32}, \texttt{bool}, S\}}{\alpha \vdash \tau \text{ bf}} \qquad \frac{\alpha \vdash \tau \text{ bf}}{\alpha \vdash \texttt{Vec<}\tau\texttt{>} \text{ bf}} \qquad \frac{\beta \vdash \tau \text{ bf} \quad \beta \geq \alpha}{\alpha \vdash \texttt{&}\beta\, m\tau \text{ bf}}$$

Devemos em particular verificar que o tipo de da sub-expressão é igualmente válido para o prazo de vida actual, *i.e.*, o do bloco no qual ele se encontra. Em Pico Rust, um empréstimo `&`me faz-se sempre para um prazo de vida igual ao do bloco actual.

A relação de ordem sobre os prazos de vida induz uma relação de ordem sobre os tipos, denotada $\tau_1 \leq \tau_2$, da seguinte forma :

$$\frac{}{\tau \leq \tau} \qquad \frac{\tau \leq \tau'}{\texttt{Vec<}\tau\texttt{>} \leq \texttt{Vec<}\tau'\texttt{>}} \qquad \frac{\tau \leq \tau' \quad \alpha \geq \beta}{\texttt{&}\alpha\, \tau \leq \texttt{&}\beta\, \tau'} \qquad \frac{\alpha \geq \beta}{\texttt{&}\alpha\, \texttt{mut}\tau \leq \texttt{&}\beta\, \texttt{mut}\tau}$$

Aquando de uma atribuição e_1=e_2, deve-se verificar se temos $\tau_2 \leq \tau_1$, onde τ_2 é o tipo de e_2 e τ_1 o tipo e_1. É preciso ter em atenção que falamos aqui dos tipos obtidos *após* eventuais coerções de `&mut` τ para `&` τ descritas mais acima.

Há verificações por fazer em vários locais, em particular

[1] Um prazo de vida pode assim ser materializada por um inteiro, que conta quantos blocos aninhados estão por baixo do bloco atual.

- quando um valor esquerdo é usado como valor direito (em particular o estatuto de uma variável pode ser modificado) ;
- aquando de uma atribuição e_1=e_2 : para além da verificação de subtipagem acima referida, é necessário verificar que o recurso que corresponde ao valor esquerdo e_1 não é um empréstimo ;
- aquando de um empréstimo, isto é para a construção `&`me. Sabemos que e é um valor esquerdo . Se e é da forma $e_1[e_2]$ ou $e_1.x$, procuramos a variável relacionada em e_1. Se e é da forma `*`e_1, poderemos então impor que e_1 seja uma variável.

2.6 Algumas pistas

Testes. No caso de dúvidas sobre questões semânticas, pode utilizar o compilador `rustc` como referência. Poderá, alias, espreitar e adotar as mensagens de erro que este compilador produz.

Anticipação. Na fase seguinte (produção de código), certas informações produzidas aquando da tipagem vão ser úteis (e utilizadas). É-vos aconselhado antecipar estas necessidades programando as funções de tipagem tais que não se limitam em varrer as árvores de sintaxe abstracta oriundas da análise sintáctica, mas sim tais que devolvam novas árvores de sintaxe abstractas enriquecidas com informações suplementares que poderão ser usadas posteriormente.

3 Restrições/diferenças em relação ao Rust

Todo o programa Pico Rust é também um programa Rust correto. A linguagem Pico Rust sofre de algumas limitações quando comparada com Rust. Em particular

- Pico Rust tem menos palavras chaves do que o Rust.
- os índices para aceder ao tipo `Vec` em Pico Rust são de tipo `i32`, enquanto são de tipo `usize` em Rust ; tal como o resultado do método `len`.

O vosso compilador não será nunca testado sobre programas incorrectos no contexto de Pico Rust (resp. Rust) mas correctos no contexto de Rust (resp. Pico Rust).

4 Produção de código

Para um ficheiro devidamente tipado, a ultima fase de um compilador consiste em produzir código x86-64 para a sua execução. O objectivo é conseguir a implementação de um compilador simples mas correto. Em particular preferiremos o uso da pilha à alocação de registos aquando do processamento dos resultados de cálculos intermédios. Claro, é possível e até desejável utilizar localmente os registos x86-64.

4.1 Semântica

O modo de passagem de Pico Rust é *por valor*, qualquer que seja o tipo. Em particular, as estruturas são atribuídas, passadas em argumento e devolvidas como resultados *por valor*, o que obriga a uma cópia de um bloco de bytes inteiro. A secção seguinte detalha o que é um valor.

Grande parte das construções (aritméticas, lógicas, condicionais, ciclos, funções) são idênticas às que encontramos em outras linguagens, em particular em C, e compilados de forma em tudo semelhante.

Um bloco pode concluir-se por uma última expressão que não se termina por ponto-e-vírgula e assim tem como valor o resultado desta mesma expressão. esta última expressão pode ser uma condicional. O corpo de uma função é um bloco e, na ausência de `return`, o valor devolvido pela função é o valor deste bloco. A construção `&`$m\,e$ designa o endereço onde se encontra na memoria o valor de e. O valor m não tem o mesmo papel aqui (somente no caso da tipagem). A construção `*`e devolve o valor que se encontra em memória no endereço designado por e.

4.2 Representação dos valores

Uma das dificuldades deste trabalho está no facto de que os dados não têm todos o mesmo tamanho ; em particular, o compilador deve saber calcular a representação memória para cada tipo.

Sugere-se a adoção, pelo menos inicialmente, da representação simples e uniforme seguinte :

- Os valores de tipo `()`, `i32`, `bool` e `&`τ estão todas representadas sobre 64 bits, da forma seguinte :
  - um valore de tipo `()` é qualquer (não é possível comparar valores de tipo `()` em Pico Rust) ;
  - um valor de tipo `i32` é um inteiro 32 bits com sinal arquivado em 64 bits (com a extensão de sinal)[2] ;
  - um valor de tipo `bool` é um inteiro 64 bits que vale 1 (para `true`) ou 0 (para `false`) ;
  - um valor de tipo `&`τ é um apontador para um valor de tipo τ.
- Um valor de tipo S, onde S é uma estrutura, é a concatenação de todos os campos de S, numa ordem deixado ao critério do compilador.
- Um valor de tipo `Vec<`τ`>` fica representado como uma estrutura possuindo dois campos, o primeiro sendo o número n de elementos e o segundo sendo um apontador para uma zona de memória alocada na *heap* de tamanho $n \times s$ bytes, onde s é o tamanho a representação de um valor de tipo τ.

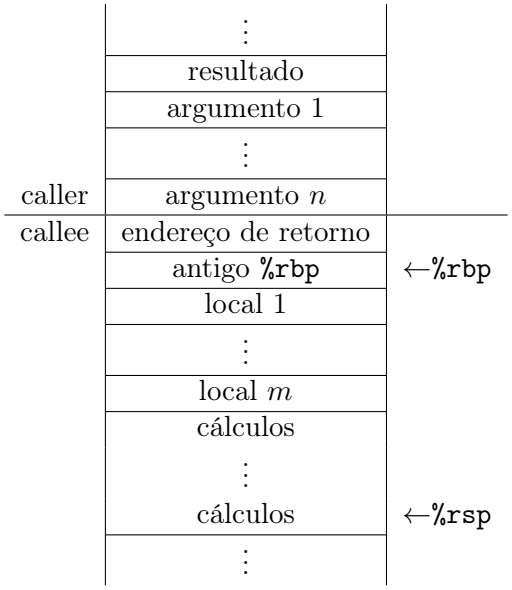
  A propósito deste último tipo, é importante perceber que a cópia de um valor de tipo `Vec<`τ`>` só obriga à cópia dos 16 bytes contendo o número de elementos e o apontador e não à cópia da zona da *heap* que contém os elementos.

4.3 Esquema da compilação

As variáveis locais são alocadas na pilha (na tabela de activação). Os argumentos e o resultado de uma função são transmitidos com base no uso da pilha (sendo os dados de tamanho variável, é o mais simples)

Poderemos adotar o esquema seguinte :

[2] Sim, parece paradoxal, mas trabalhar exclusivamente com registos 64 bits simplificará em muito a produção de código e, honestamente, não é incorrecto representar um inteiro 32 bits por um inteiro 64 bits. Pode-se neste caso ignorar os problemas de *artihmetic overflow* nos testes fornecidos e usados.



Para a cópia de estruturas (passagem de argumentos, valor de retorno, atribuição), poder-se-á escrever uma função do estilo `memmove`, por exemplo directamente em x86-64. A função predefinida `print!` poderá ser concretizada por uma chamada a função da biblioteca `printf` (com os bons argumentos).

Sugestão de organização. Poderá proceder conforme as etapas seguintes :

1. calcular a representação (localização dos campos e tamanho total) de todas as estruturas ;
2. para cada função f do código :
   (a) calcular a localização dos seus argumentos/resultado na pilha,
   (b) calcular a localização das suas variáveis locais na pilha,
   (c) produzir o código x86-64 da função f ;
3. juntar o código de `print!` e das funções auxiliares, caso necessário ;
4. alocar as strings no segmento de dados.

5 Trabalho Requerido

O trabalho pode ser realizado em grupo de dois até quatro elementos. Deve ser enviado por email a `desousa@di.ubi.pt` com o Assunto: "DLPC: Entrega de Trabalho".

O trabalho deve ser entrega na forma de um arquivo `tar` comprimido (a opção "`z`" de `tar`), de nome *vossos_nomes*`.tgz` e deve conter uma pasta co o nome *vossos_nomes* (exemplo : `joaoalmeida_antonioveiga.tgz`). Nesta pasta devemos encomntrar as *fontes* do vosso programa (é inútil juntar os ficheiros compilados, estes srrão gerados). Quando se está nesta pasta, o comando `make` deve criar o vosso compilador, que terá por nome `prustc`. O comando `make clean` deve apagar todos os ficheiros que `make` gerou e deixar na pasta exclusivamente os ficheiros fontes.

O arquivo deve igualmente conter um curto relatório explicando as diferentes escolhas técnicas que foram tomadas e, caso necessário, as dificuldades encontradas ou os elementos do trabalho pedido que não foram implementadas. Este relatório poderá estar na forma ASCII, Markdown, PostScript ou PDF.

O vosso compilador deve aceitar em opção de linha de comando um só ficheiros Pico Rust (de extensão `.rs`) e eventualmente uma opção desta lista: `--parse-only`, `--type-only` e `--no-asm`. Estas opções indicam respectivamente de parara a compilação após a análise sintáctica (secção 1), a tipagem simples (até 2.3) ou mesmo antes da produção de código (logo, com a tipagem de recursos realizada).

Em caso de erro léxico, sintáctico ou de tipagem, este deve ser assinalado (da forma abaixo indicada) e o programa deve terminar com o código de saída 1 (`exit 1`). Em caso de erro diferente (um erro do próprio compilador, por exemplo), o programa deve terminar co o código de saída 2 (`exit 2`).

Quando um erro é detectado pelo vosso compilador, este deve ser assinalado com a maior precisão possível, pela sua natureza e a sua localização no ficheiro fonte do programa compilado. Adotaremos o formato seguinte para assinalar os erros :

```
File "test.rs", line 4, characters 5-6:
syntax error
```

O formato é útil para os IDE como emacs que lhes permite que funções como `next-error` (de emacs) possam interpretar a localização e colocar o cursor automaticamente no local de erro assinalado. Poderão, se o pretenderem, apresentar uma mensagem de explicação do erro na forma que mais vos convier. As localizações de erro podem ser obtidas durante a análise sintáctica graças as palavras chaves `$startpos` e `$endpos` de Menhir, e conservá-las, se necessário, na árvore de sintaxe abstracta.

Se o ficheiro de entrada está conforme a sintaxe e a politica de tipos descrita neste documento, o vosso compilador produzirá código x86-64 e terminará com o código de saída 0 (`exit 0` explícito ou fim normal de programa), sem produzir nenhum output na saída standard.

Se o ficheiro de entrada for `file.rs`, o código x86-64 deve ser produzido no ficheiro `file.s` (com o mesmo nome de ficheiro que o ficheiro fonte, mas com a extensão `.s` no lugar de `.rs`). Este ficheiro x86-64 deve poder ser compilado e executado com os comandos

```
gcc file.s -o file
./file
```

O resultado produzido na saída standard deve ser idêntico ao resultado produzido por

```
rustc file.rs
./file
```

Agradecimentos. Este trabalho e o seu enunciado são fruto de adaptações directas de um trabalho produzido pelo Jean-Christophe Filliâtre que agradeço pessoalmente. No trabalho original, refere-se que este também contou com o contributo de Jacques-Henri Jourdan. Muito obrigado aos dois.