Eparch: Máquinas de Estado Estaticamente Tipadas em Gleam

Marcos Benevides
Rolim de Moura, Brasil
marcos.schonfinkel@gmail.com

Fernando Areias
Departamento Acadêmico de Informática (DAINF), Universidade Tecnológica Federal do Paraná (UTFPR)
Curitiba, Brasil
fernandoareias@alunos.utfpr.edu.br

Adolfo Neto
Departamento Acadêmico de Informática (DAINF), Universidade Tecnológica Federal do Paraná (UTFPR)
Curitiba, Brasil
adolfo@utfpr.edu.br

Resumo

O ecossistema Erlang/OTP e sua máquina virtual BEAM são referência para sistemas distribuídos e tolerantes a falhas. Nos últimos anos, surgiram linguagens com tipagem estática sobre a BEAM, como Gleam, que buscam detectar erros em tempo de compilação. Contudo, os comportamentos genéricos do OTP, como gen_statem e gen_event, são inerentemente dinâmicos, permitindo erros de lógica que só se manifestam em tempo de execução. Este trabalho apresenta a Eparch, uma biblioteca em Gleam que fornece um *wrapper* estaticamente tipado sobre esses módulos, permitindo que transições de estado e tratamento de mensagens sejam verificados pelo compilador. A biblioteca utiliza Tipos de Dados Algébricos e *pattern matching* exaustivo para modelar estados, eventos e transições, adotando o padrão Builder funcional para oferecer uma API declarativa e composicional. A Eparch cobre praticamente todas as funcionalidades modernas do gen_statem, eliminando em tempo de compilação uma classe inteira de erros de *runtime* comuns em implementações puramente dinâmicas, sem sacrificar o acesso ao modelo de concorrência e supervisão do OTP. Como trabalhos futuros, os autores apontam a implementação de composição intercalada com possível uso de *Session Types*, além da exploração de meta-programação para automatizar a camada de FFI.

Palavras-chave

Gleam, Erlang, BEAM, Tipagem Estática, Máquinas de Estado.

1 Introdução

Erlang é uma linguagem de programação funcional projetada para atender aos requisitos de alta disponibilidade dos sistemas de telecomunicações da Ericsson, os quais demandavam elevada resiliência e tolerância a falhas. Em decorrência desses requisitos, sua máquina virtual, a BEAM (*Bogdan/Björn's Erlang Abstract Machine*), consolidou-se como uma plataforma de referência para o desenvolvimento de sistemas distribuídos, principalmente em razão de seu modelo de concorrência e do *framework* OTP (*Open Telecom Platform*) [1, 2].

Nos últimos anos, a evolução do ecossistema da BEAM deu origem a diversas investigações sobre tipagem gradual ou verificação estática de código Erlang/Elixir, tais como o `dialyzer`[1] ou a introdução de *Set-Theoretic Types* no sistema de tipos da linguagem Elixir ([4]). Além dessas duas últimas abordagens, temos o surgimento de linguagens com sistemas de tipos estáticos e rigorosos, como Gleam[2], que buscam detectar erros ainda em tempo de compilação. Nesse contexto, o principal desafio reside na integração desses sistemas de tipos com os comportamentos dinâmicos e genéricos oferecidos pelo OTP, como o `gen_statem`[3] e o `gen_event`[4], componentes fundamentais para a implementação de máquinas de estados complexas e de sistemas orientados a eventos.

Diante desse desafio, este trabalho apresenta a Eparch[5], uma biblioteca desenvolvida em Gleam que oferece abstrações tipadas sobre os módulos `gen_statem` e `gen_event` do OTP. A Eparch aborda diretamente o problema central identificado: a incompatibilidade entre a natureza dinâmica desses comportamentos genéricos e as garantias estáticas proporcionadas pela linguagem Gleam. Por meio de uma API idiomática, a biblioteca permite que desenvolvedores definam máquinas de estados e manipuladores de eventos de forma declarativa, delegando ao compilador a verificação de que as mensagens trocadas entre processos, os estados alcançados e as transições realizadas estejam em conformidade com os tipos definidos. Dessa forma, uma classe inteira de erros de *runtime*, frequentes em implementações puramente dinâmicas, é eliminada antes mesmo da execução do programa, sem sacrificar o acesso ao modelo de concorrência e aos mecanismos de supervisão de falhas que tornam o OTP uma referência no desenvolvimento de sistemas de alta disponibilidade. A criação desta biblioteca foi motivada justamente pela ausência de suporte ao `gen_statem` na biblioteca OTP oficial da linguagem Gleam (a gleam/otp).

2 Eparch

A biblioteca Eparch utiliza o sistema de tipos da linguagem Gleam para formalizar essas definições, garantindo que cada par (estado, evento) corresponda a uma transição válida e tipada. Dessa forma, elimina, ainda em tempo de compilação, a possibilidade de o processo alcançar estados indefinidos ou processar mensagens incompatíveis com o contexto de execução atual.

2.1 O padrão `Builder`

Em Gleam, o padrão *Builder* constitui uma solução funcional elegante que explora o uso de *pipelines* (por meio do operador `|>`) e de registros (*records*) imutáveis para permitir o desenvolvimento de APIs (Interfaces de Programação de Aplicações) intuitivas e expressivas. Nessa abordagem, a API fornece um registro de configuração inicial, juntamente com um conjunto de funções de transformação responsáveis por atualizar campos específicos, convertendo uma

[1] https://www.erlang.org/doc/apps/dialyzer/dialyzer.html

[2] https://gleam.run/

[3] https://www.erlang.org/doc/apps/stdlib/gen_statem.html

[4] https://www.erlang.org/doc/apps/stdlib/gen_event.html

[5] https://github.com/byzantine-systems/eparch

lógica de configuração potencialmente complexa em uma sequência de operações clara, legível e composicional.

O Código 1 ilustra essa abordagem na prática: a função `start_with_lock_timeout` recebe um código numérico e um tempo limite de travamento automático, e constrói uma máquina de estados por meio de um *pipeline*. Primeiro, `sm.new` cria o registro inicial com o estado `Locked` e os dados de configuração. Em seguida, `sm.on_event` registra o tratador de eventos, `sm.with_state_enter` habilita a execução de lógica na entrada de cada estado e, por fim, `sm.start` inicia a máquina. Cada etapa recebe o registro da anterior e devolve uma versão atualizada, sem qualquer mutação.

Código 1: Habilitando funcionalidades de forma funcional

```
pub fn start_with_lock_timeout(
  code: List(Int),
  auto_lock_ms: Int,
) -> sm.StartResult(Message) {
  sm.new(initial_state: Locked, initial_data: Data
      (code: code, remaining: code))
  |> sm.on_event(fn(event, state, data) {
    handle_event(auto_lock_ms, event, state, data)
  })
  |> sm.with_state_enter()
  |> sm.start
}
```

Essa abordagem é uma "maneira funcional" de evitar os temidos problemas de "sopa de construtor" e, ao mesmo tempo, garantir que a *API* permaneça segura e detectável: os usuários têm a satisfação de "construir" uma configuração sem o risco de alterar acidentalmente o estado ou passar parâmetros anônimos na ordem errada, tornando a biblioteca mais agradável de usar e robusta por padrão.

2.2 O Sistema de Tipos

Para demonstrar o uso da biblioteca, traduzimos para Gleam o exemplo *pushbutton*, disponível na documentação oficial do `gen_statem`[6], utilizando Tipos de Dados Algébricos (*ADTs*) para garantir, em tempo de compilação, que transições inválidas sejam impossíveis.

O exemplo modela um botão com dois estados, `Off` e `On`, cujas transições são disparadas pelo evento `push`. A cada transição de `Off` para `On`, um contador é incrementado. A Figura 1 apresenta o diagrama de estados correspondente.

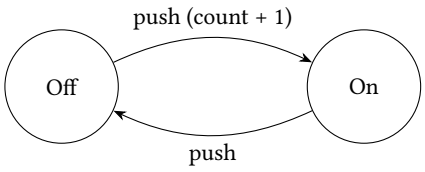

Figura 1: Diagrama do exemplo oficial da documentação do `gen_statem`.

A tradução para Gleam começa pela definição dos tipos que estruturam a máquina de estados, conforme o Código 2. O tipo `State` enumera os estados possíveis (`Off` e `On`); `Data` encapsula o contador de acionamentos; `Message` define os eventos que a máquina pode receber (`Push` para alternar o estado e `GetCount` para consultar o contador); e `Reply` representa as respostas possíveis, informando o novo estado ou o valor atual do contador.

Código 2: Definição dos tipos do exemplo *pushbutton* em Gleam

```
pub type State { Off On }
pub type Data { Data(count: Int) }
pub type Message { Push GetCount }
pub type Reply {
  PushReply(state: State)
  CountReply(count: Int)
}
```

A lógica de transições é definida através de uma função pura (`handle_event`), onde o compilador verifica a exaustividade dos estados. O uso do *pattern matching* elimina a necessidade de condicionais complexas: qualquer combinação não tratada explicitamente é ignorada e não altera o estado da máquina de estados.

O Código 3 apresenta essa função. O casamento de padrões opera simultaneamente sobre o evento recebido e o estado atual: quando o evento é uma chamada (`sm.Call`) com a mensagem `Push` e a máquina está em `Off`, ocorre a transição para `On` com o contador incrementado, quando está em `On`, retorna para `Off` sem alterar o contador. Para a mensagem `GetCount`, independentemente do estado, a máquina permanece no estado atual e responde com o valor do contador. O último ramo (`_, _`) captura qualquer outra combinação, mantendo o estado inalterado.

Código 3: Tratamento de eventos com *pattern matching* sobre chamadas do tipo `Call` para os eventos `Push` e `GetCount`

```
pub fn handle_event(
  event: sm.Event(State, Message, Reply),
  state: State,
  data: Data,
) -> sm.Step(State, Data, Message, Reply) {
  case event, state {
    sm.Call(from, Push), Off ->
      sm.next_state(On, Data(count: data.count +
          1), [
        sm.reply(from, PushReply(On)),
      ])
    sm.Call(from, Push), On ->
      sm.next_state(Off, data, [sm.reply(from,
          PushReply(Off))])
    sm.Call(from, GetCount), _ ->
      sm.keep_state(data, [sm.reply(from,
          CountReply(data.count))])
    _, _ -> sm.keep_state(data, [])
  }
}
```

2.3 Limitações

Cruzar a fronteira entre o sistema de tipos estrito e estático do Gleam e a filosofia dinâmica do Erlang, que admite falhas em tempo

[6] https://www.erlang.org/doc/apps/stdlib/gen_statem.html#module-pushbutton-code

de execução, introduz um conjunto singular de desafios. Ao definir uma função `@external`, estabelece-se um contrato com o compilador acerca dos tipos de dados que o código Erlang aceitará e retornará, cabendo ao compilador do Gleam assumir que esse contrato será respeitado. Caso a implementação em Erlang viole essa expectativa e se comporte de maneira incompatível com a interface declarada, as garantias oferecidas pelo sistema de tipos deixam de ser válidas, possibilitando a introdução de falhas em tempo de execução.

Quando não há certeza absoluta quanto aos tipos que serão retornados, é possível utilizar o módulo `gleam/dynamic` para decodificar os dados com segurança em tempo de execução. Essa abordagem converte possíveis falhas decorrentes da interação com a BEAM em valores do tipo `Result(T, List(DecodeError))`, permitindo que sejam tratadas de forma explícita e segura pelo programador. Ainda assim, a estratégia mais adequada para lidar com a natureza dinâmica do Erlang e minimizar a necessidade de recorrer ao tipo `Dynamic` consiste em modelar o problema de modo que os comportamentos dinâmicos permaneçam isolados nas fronteiras do sistema e sejam representados, sempre que possível, por meio do próprio sistema de tipos de Gleam.

Mesmo possuindo um sistema de tipos robusto e inspirado por linguagens da família ML (o que já é um ganho de ergonomia quando comparado a linguagens dinâmicas da *BEAM*), uma das grandes dificuldades durante o design de nossa biblioteca foi a ausência de recursos avançados que alavancassem ainda mais o sistema de tipos. *Gleam* é deliberadamente muito mais simples que linguagens como *OCaml* ou *Standard ML*, seus módulos funcionam principalmente como um mecanismo de *namespaces* e unidades de compilação, enquanto seu sistema de tipos deriva a maior parte de seu poder de abstração com o uso de tipos paramétricos (via *Phantom* ou *Opaque Types*).

Nesse contexto, os autores sentiram limitações significativas decorrentes da ausência de recursos como *macros* ou um sistema de módulos de primeira classe/parametrizável (ao estilo dos *Functors* em *OCaml*). A inclusão de qualquer um desses recursos (ou mesmo de uma combinação deles) beneficiaria imensamente nosso experimento, permitindo transferir grande parte do design e da verificação da biblioteca para o sistema de tipos.

3 Conclusão

Nossa biblioteca demonstra que é possível usufruir do ecossistema da BEAM sem abrir mão das garantias oferecidas pelo sistema de tipos do Gleam, transformando interações potencialmente inseguras com o Erlang em fluxos de trabalho previsíveis e verificáveis pelo compilador. Dessa forma, estabelece uma base sólida para que desenvolvedores possam explorar o potencial da BEAM com maiores garantias de corretude, reforçando o papel da linguagem Gleam como uma camada de abstração mais ergonômica para o desenvolvimento de sistemas distribuídos resilientes. Nossa biblioteca oferece suporte para praticamente todos os recursos modernos do `gen_statem` (vide a tabela 1).

3.1 Trabalhos Futuros

Um desenvolvimento futuro pode ser direcionado à implementação em Gleam do conceito de composição intercalada (*interleaving composition*), originalmente proposto por Bocchi, Orchard e Voinea [3] sobre o comportamento `gen_statem` do Erlang/OTP. Uma implementação em *Gleam* teria o potencial de tornar toda a álgebra de processos (*process algebra*) ergonômica ao nível do sistema de tipos, possivelmente por meio de *Session Types*, originalmente introduzidos Takeuchi, Honda e Kubo [9]. Quanto à implementação de *Session Types* em linguagens tipadas, o trabalho de Jespersen, Munksgaard e Larsen [7] investiga como modelar *Session Types* em *Rust*, incluindo um exemplo interessante com uma modelagem de um caixa eletrônico usando este formalismo. O exemplo usado foi originalmente formulado por Honda, Vasconcelos, Kubo [6] e sua modelagem em *Gleam* serviria como um ótimo comparativo entre os pontos fortes (e limitações) das duas linguagens.

Outro eixo de investigação pode ser baseado no trabalho de Rossberg, Russo e Dreye [8], que poderia (em tese) ser usado para implementar um sistema de *Module Functors* mais restrito, de uma forma semelhante ao que acontece na linguagem *Sesterl*[7]. Esse tipo de abstração já seria bem mais poderosa do que qualquer outro recurso suportado pela linguagem atualmente. Por último, podemos explorar o suporte de *Gleam* à meta-programação, o que poderia eliminar o ônus de manter manualmente a camada de FFI (*Foreign Function Interface*) em Erlang, ou até mesmo derivar automaticamente os duals em uma futura implementação de *Session Types*.

Tabela 1: Comparação entre recursos modernos do `gen_statem`, a versão do *OTP* onde foram introduzidos e sua *API* equivalente na biblioteca `eparch`

OTP	Recurso	(API) eparch
19.2	State enter calls	`with_state_enter`, evento `Enter`
20.0	Named generic timeouts	`generic_timeout`, tipo `GenericTimeout`
20.0	Idle hibernation	`with_hibernate_after`
22.1	Explicit timeout cancellation	`cancel_{state,event,generic}_timeout`
22.1	Timeout content update	`update_{state,event,generic}_timeout`
22.3	Runtime callback module swap	`change_callback_module`
22.3	Callback module stack	`push_callback_module`, `pop_callback_module`
23.0	Atomic start plus monitor	`start_monitor`, `MonitoredMachine`
23.0	Asynchronous calls	`send_request`, `RequestId(reply)`
23.0	Blocking and bounded waits	`wait_response`, `wait_response_timeout`
23.0	Non-blocking response check	`check_response`
24.0	Selective response receive	`receive_response`, `receive_response_blocking`
25.0	Request-id collections	`request_ids_{new,add,size,to_list}`
25.0	Batched async calls	`send_request_to_collection`, `receive_response_collection`
25.0	Map-based status formatting	`on_format_status`, `Status`

Disponibilidade de Artefatos

O projeto está disponível publicamente no GitHub em https://github.com/byzantine-systems/eparch sob a licença LGPL. O ambiente de desenvolvimento e toda a automatização dos testes locais (e em CI) assumem o uso de Nix [5].

[7] https://github.com/gfngfn/Sesterl

Referências

[1] Joe Armstrong. 2003. *Making reliable distributed systems in the presence of software errors.* Ph. D. Dissertation.

[2] Joe Armstrong. 2007. A history of Erlang. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages* (San Diego, California) *(HOPL III)*. Association for Computing Machinery, New York, NY, USA, 6–1–6–26. doi:10.1145/1238844.1238850

[3] Laura Bocchi, Dominic Orchard, and A Laura Voinea. 2022. A theory of composing protocols. *Art, Science, and Engineering of Programming* 7, 2 (2022).

[4] Giuseppe Castagna, Guillaume Duboc, and José Valim. 2023. The design principles of the elixir type system. *arXiv preprint arXiv:2306.06391* (2023).

[5] Eelco Dolstra, Merijn De Jonge, Eelco Visser, et al. 2004. Nix: A Safe and Policy-Free System for Software Deployment.. In *LISA*, Vol. 4. 79–92.

[6] Kohei Honda, Vasco T Vasconcelos, and Makoto Kubo. 1998. Language primitives and type discipline for structured communication-based programming. In *European Symposium on Programming*. Springer, 122–138.

[7] Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. 2015. Session types for Rust. In *Proceedings of the 11th acm sigplan workshop on generic programming*. 13–22.

[8] Andreas Rossberg, Claudio Russo, and Derek Dreyer. 2014. F-ing modules. *Journal of functional programming* 24, 5 (2014), 529–607.

[9] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. 1994. An interaction-based language and its typing system. In *International Conference on Parallel Architectures and Languages Europe*. Springer, 398–413.