Sistema de Suporte à Execução de uma Linguagem Web Reativa

João Mateus, Miguel Domingues and João Costa Seco

Faculdade de Ciências e Tecnologia - Universidade Nova de Lisboa

Resumo O desenvolvimento de *software* segundo metodologias ágeis é dividido num conjunto de etapas que são sucessivamente repetidas, onde se produzem estados intermédios dos sistemas a desenvolver. No caso das aplicações centradas em dados persistentes, como as aplicações *web* e *cloud*, o uso destas metodologias resulta na evolução contínua do código e respetivo esquema de dados. Devido às constantes modificações, a probabilidade de introdução de erros de programação aumenta, principalmente quando é necessário manter a consistência entre código e dados. A uma escala mais pequena, as tarefas de programação, compilação, teste e *deployment* também prejudicam a produtividade do programador, pois adiam para o fim do ciclo informação relevante sobre o funcionamento da aplicação.
Neste artigo propomos uma solução baseada numa linguagem de programação e num ambiente de desenvolvimento *live*, que fornece *feedback* constante e imediato quanto ao estado atual da aplicação (código e dados). O sistema garante, estaticamente, que todas as modificações efetuadas à aplicação são isentas de erros. O sistema suporta ainda, concorrentemente, a evolução e execução de aplicações centradas em dados persistentes.

Keywords: Programação *live*, evolução de *software*, linguagem reativa de *data-flow*, aplicações *web*

1 Introdução

O desenvolvimento de aplicações segundo metodologias ágeis segue um ciclo de desenvolvimento composto por diversas etapas (escrita de código, compilação, teste, *deployment*). Este ciclo é iterado múltiplas vezes, adicionando incrementalmente funcionalidades à aplicação. Em particular, no caso de aplicações *web*, o ciclo de desenvolvimento divide-se em: especificação, implementação, compilação, teste e *deployment*. Só depois de concluída a primeira iteração, e respetivo primeiro *deployment*, é possível utilizar as funcionalidades especificadas. As diversas etapas desde a implementação da aplicação até à sua execução no servidor impedem o programador de ter, de modo imediato, uma noção clara de como a aplicação funciona e se cumpre a especificação inicial. O processo de compilação e *deployment* distanciam o programador da execução da aplicação, o que pode prejudicar a sua produtividade, ao atrasar *feedback* potencialmente valioso sobre

o comportamento da aplicação. Além disso, durante o processo de evolução de aplicações centradas em dados, também pode haver a introdução de novos *bugs*, derivados da evolução colateral do código e do respetivo esquema de dados [8]. A utilização de uma ferramenta de programação que fornece *feedback* imediato ajuda a diminuir a distância entre a implementação e a execução, permitindo ao programador implementar e utilizar a aplicação em simultâneo. Neste artigo, apresenta-se um ambiente de desenvolvimento *live* que permite visualizar o efeito que cada modificação tem no funcionamento da aplicação, aumentando assim a produtividade e eficiência do processo de desenvolvimento [3,10]. Para além do *feedback*, a reconfiguração incremental do sistema sem a necessidade de o interromper ou reiniciar tem a vantagem de evitar quebras de serviço, mantendo as aplicações sempre disponíveis.

A nossa abordagem consiste na extensão e implementação de uma linguagem incremental e reativa, que permite desenvolver aplicações *web* centradas em dados persistentes de um modo incremental e mantendo o estado da aplicação sempre atualizado [2]. Cada um dos incrementos é verificado estaticamente pelo sistema de tipos da linguagem, garantindo que a aplicação evolui de forma segura. Após a verificação estática de cada incremento, este é introduzido na aplicação em execução, sendo que o *deployment* é feito de forma transparente e imediata, sem qualquer interrupção ou disrupção do sistema. Desta forma, elimina-se a necessidade de duas etapas distintas para compilação e *deployment* [1].

Para fornecer *feedback* imediato, a aplicação precisa de refletir os dados e o código mais recentes. Uma solução passa por conceder propriedades reativas à aplicação, que garantem a propagação de todas as alterações efetuadas. A reatividade suportada pela linguagem tem por base um grafo de dependências entre os vários elementos da aplicação. Sempre que um destes elementos é modificado, essa alteração é propagada pelo grafo, atualizando todos os elementos cujas dependências tenham sofrido alterações. Desta forma, o sistema garante que todos os elementos estão sincronizados entre si e que a aplicação reflete o código e dados que estão presentes no sistema. De modo a evitar que a propagação das alterações entre num ciclo infinito, é necessário que o grafo de dependências seja acíclico, algo que é garantido estaticamente pelo sistema de tipos. Ao combinar o desenvolvimento incremental com a reatividade, conseguimos fornecer ao programador um ambiente de desenvolvimento *live*, onde todas as alterações efetuadas ao sistema podem ser vistas e utilizadas em tempo-real na aplicação em execução, melhorando a experiência de desenvolvimento do programador.

No contexto deste projeto, foi desenvolvido um protótipo [9] onde se representam os conceitos anteriores. O sistema presente neste protótipo fornece um ambiente de desenvolvimento (IDE), acessível através do *browser*, onde se pode programar o comportamento e a interface de uma aplicação *web*, combinado na mesma linguagem o *front-end* (interface HTML) e o *back-end* (lógica e esquema de dados). A semântica da linguagem é mapeada para elementos de uma aplicação *web*. Por exemplo, uma função que devolve um documento HTML é convertida para uma página HTML com parâmetros. Para propagar as alterações pelo grafo de dependências, o sistema utiliza uma fila de operações onde são adicionados os

nomes dos elementos que necessitam de ser atualizados. Os clientes (*browsers*) utilizam o padrão do observador em relação aos elementos de uma página. Sempre que o valor associado a um elemento muda, é enviada uma mensagem via WebSockets para as páginas (clientes) que estão, nesse momento, subscritas a esse nome, indicando qual o novo valor.

As principais contribuições deste projeto são:

- Uma linguagem de programação, orientada ao desenvolvimento de aplicações *web* reativas, permitindo implementar a lógica, esquema de dados e interface;
- Um sistema que possibilita o desenvolvimento de aplicações *web* de modo incremental, garantindo estaticamente que a evolução do sistema não causa qualquer disrupção;
- Um ambiente de desenvolvimento *live* que oferece ao programador *feedback* imediato e contínuo quanto ao seu código, conferindo propriedades reativas às aplicações que mantêm a interface sincronizada com o código e dados.

Na Secção 2 apresenta-se detalhadamente a linguagem através de um exemplo construido incrementalmente. De seguida, a Secção 3 descreve o sistema de suporte à execução, e as Secções 4 e 5 detalham o trabalho relacionado e trabalho futuro, respetivamente. Por fim, na Secção 6 é feita uma discussão final do projeto.

2 Linguagem

O foco da linguagem aqui apresentada é possibilitar o desenvolvimento dos aspetos base de uma aplicação *web* reativa centrada em dados persistentes, i.e., lógica, esquema de dados e interface HTML. A linguagem suporta desenvolvimento incremental, onde cada incremento é verificado estaticamente pelo sistema de tipos e introduzido na aplicação sem qualquer disrupção, garantindo a evolução segura e contínua da aplicação. Recorrendo a um grafo de dependências, a linguagem também possui propriedades reativas, mantendo o código e dados de uma aplicação permanentemente sincronizados. Esta reatividade permite manter as interfaces sempre atualizadas, i.e., sempre que os dados ou o código mudam, as interfaces são refrescadas para refletirem o novo estado da aplicação. Deste modo, o programador recebe *feedback* imediato do sistema, sabendo sempre qual o impacto do seu código na aplicação em execução.

A base da linguagem consiste em três operações básicas: `var`, `def` e `do`. A operação `var` é utilizada para declarar nomes que armazenem o estado da aplicação. A operação `def` associa uma transformação pura de dados a um nome. Por último, a operação `do` permite executar ações (definidas por intermédio da operação `action`), que por sua vez alteram o estado da aplicação.

De seguida, é exemplificada a criação de uma pequena aplicação *web* que demonstra algumas das capacidades da linguagem. O objetivo principal desta aplicação é a criação de um contador (lógica), que pode ser incrementado várias vezes. Além disso, pretende-se criar também uma página HTML (interface) com o intuito de apresentar o valor atual do contador, juntamente com um botão que permite incrementar o valor do contador. Por fim, serão apresentadas algumas

```
var counter = 0
var inc = action { counter := counter + 1 }
var plus5 = counter + 5
def times2 = counter * 2
```

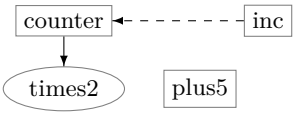

Figura 1. Exemplo de um contador.

reconfigurações à aplicação para acrescentar ou modificar funcionalidades, como mostrar outros valores ou definir manualmente o valor do contador.

A Figura 1 ilustra a criação do contador, recorrendo a um número inteiro. Na primeira linha do exemplo é definida uma variável de estado (`var`) `counter`, com o propósito de armazenar o valor (número inteiro) do contador. O nome `inc` denota uma ação que incrementa o contador (`counter`) numa unidade sempre que esta é executada. Note que, ao definir o nome `inc`, a ação fica definida mas a sua execução fica suspensa, podendo mais tarde ser executada explicitamente através da operação `do`. Com estes dois nomes fica definido o núcleo da aplicação. Considere-se, como exemplo, os nomes `times2` e `plus5` definidos na Figura 1, que guardam, respetivamente, o resultado da multiplicação da variável de estado `counter` por 2 e o resultado da soma entre o valor do nome `counter` e 5. Ao definir o nome `times2` como uma transformação pura de dados (`def`), este fica dependente da variável de estado `counter` e é recalculado (atualizado) sempre que o valor de `counter` muda. Este efeito é controlado através de um grafo de dependências entre os vários nomes. Através deste grafo é possível controlar quais os nomes (transformações puras de dados – `def`) que necessitam de ser atualizados, sempre que uma das suas dependências é modificada. Desta forma, sempre que uma das dependências de um nome muda, este é recalculado, de forma a refletir o estado mais atual do sistema.

Na Figura 1 ilustra-se o grafo das dependências correspondente ao contador, presente na mesma figura. Os nós retangulares da figura (variáveis de estado – `var`), que representam o estado persistente da aplicação, não são afetados pela propagação das alterações. Já os nós elípticos (transformações puras de dados – `def`), que representam a lógica da aplicação, são recalculados sempre que, pelo menos, uma das suas dependências muda. As setas contínuas representam uma dependência entre dois nomes, apontando no sentido da propagação. As setas a tracejado representam os nomes afetados pela execução de uma ação (através da operação `do`), não sendo consideradas na propagação de alterações.

Recorrendo à operação `do`, é possível executar a ação associada ao nome `inc` (`do inc`), modificando o valor associado ao nome `counter` de 0 para 1. Como o nome `times2` representa uma transformação pura de dados (`def`) que depende do nome `counter`, o seu valor é recalculado, passando a armazenar o valor 2. Contudo, `plus5` mantém-se inalterado, porque, sendo uma variável de estado, não é afetado pela propagação das alterações no grafo de dependências, continuando a armazenar o mesmo valor (5). Se a ação `inc` fosse novamente executada, o mesmo processo seria repetido, sendo que `counter` ficaria com o valor 2, `times2` ficaria com 4 e `plus5` manter-se-ia mais uma vez inalterado (5).

```
def counterPage =
  <div>
    <h1>"Counter"</h1>
    <p>counter</p>
    <button doaction=inc>
      "Increment"
    </button>
  </div>
```

Figura 2. Página HTML do contador com um botão.

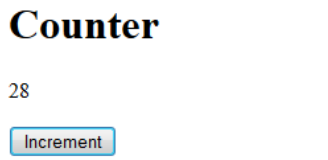

Figura 3. HTML gerado pelo código da Figura 2.

```
def counterPage step =
    <div>
        <h1>"Counter"</h1>
        <p>counter</p>
        <button doaction=(action { counter := counter + step })>
            ("Add " ++ str step)
        </button>
    </div>
```

Figura 4. Página HTML do contador com um parâmetro.

Utilizando apenas as operações básicas apresentadas (`def`, `var` e `do`), é possível definir todos os aspetos de uma aplicação *web*, incluindo as páginas HTML. A Figura 2 ilustra um exemplo de como criar uma página HTML, denominada `counterPage`, que apresenta o valor atual do `counter`. O sistema trata os nomes que devolvem documentos HTML como *routes*. Assim, é possível aceder a esta página através do URL `/counterPage`.

Para permitir a interação do utilizador com o sistema, é possível associar ações a botões. Para tal, utiliza-se o atributo `doaction` do elemento `button`. A página `counterPage`, na Figura 2, contém um botão associado à ação `inc`. Sempre que o utilizador pressiona o botão, a respetiva ação `inc` é executada, alterando o `counter`. A página HTML gerada a partir do `counterPage` está representada na Figura 3.

As páginas HTML também seguem o modelo de propagação de alterações. Desta forma, sempre que uma página é redefinida ou uma das suas dependências é alterada, as alterações são propagadas para todos os utilizadores, garantindo que a página utilizada reflete os valores atuais do sistema. Por exemplo, caso o valor do contador seja alterado, o valor exibido na página (Figura 3) é atualizado sem a necessidade de refrescar toda a página.

A linguagem suporta ainda a criação de páginas que recebam parâmetros, tal como apresentado na Figura 4. Para visualizar à nova página `counterPage` basta aceder ao URL `/counterPage/<arg>`, onde `<arg>` será o número que ficará associado ao parâmetro `step`.

Comparativamente ao `counterPage` apresentado na Figura 2, o `counterPage` da Figura 4 é muito semelhante. Contudo há duas diferenças fundamentais. A

```
def counterPage =
    <div>
        <h1>"Counter"</h1>
        <p>"Counter: " counter</p>
        <button doaction=(inc)>
            "Increment"
        </button>
        <br/>
        <br/>
        "New value: " <input type="number" id="newCounter" value=counter/>
        <button doaction=(action { counter := #newCounter })>
            "Set counter"
        </button>
    </div>
```

Figura 5. Página HTML do contador com um *input*.

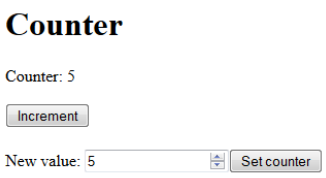

Figura 6. HTML gerado pelo código da Figura 5.

primeira está presente na primeira linha: o argumento `step`. Isto significa que `counterPage` não é apenas um valor HTML, mas sim uma função que que recebe um argumento (inteiro) e devolve uma página HTML. A outra diferença está na ação associada ao botão, que utiliza o valor passado como argumento para incrementar o `counter`. Tal como o `counterPage` definido anteriormente, o novo `counterPage` também é mantido sempre atualizado de forma a refletir os valores mais recentes das suas dependências. Por exemplo, se o utilizador aceder ao URL `/counterPage/10`, a página irá conter um botão onde se lê `"Add 10"`. Sempre que o utilizador premir o botão, o `counter` será incrementado em 10 unidades.

O mesmo exemplo poderia ser recriado recorrendo a um *input* fornecido pelo utilizador. A linguagem suporta a utilização de *inputs* HTML dentro de ações, como exemplificado na Figura 5, cuja interface HTML está representada na Figura 6. Como se pode observar, é utilizado um elemento *input* com o identificador `newCounter`. A ligação entre *inputs* na página e nomes na aplicação é feita pelo prefixo `#` (e.g. `#newCounter`). A utilização destes identificadores está restringida ao corpo das ações, pois só estas podem ter a sua execução adiada. É de realçar que, tal como o resto da linguagem, os *inputs* também são tipificados. No exemplo, o *input* `#newCounter` tem de ser do mesmo tipo do `counter`, sendo necessário adicionar o atributo `type` com a *string* `"number"`.

Na Figura 7 está representada uma nova reconfiguração do contador. A reconfiguração tem por objetivos apresentar no HTML os resultados de `plus5` e `times2`, e aumentar o incremento do contador para 2. Para tal, o nome `counterPage` é

```
var inc = action { counter := counter + 2 }

def plus5 = counter + 5

def counterPage =
    <div>
        <h1>"Counter"</h1>
        <p>"counter = " counter</p>
        <p>"counter + 5 = " plus5</p>
        <p>"counter * 2 = " times2</p>
        <button doaction=inc>
            "Increment"
        </button>
    </div>
```

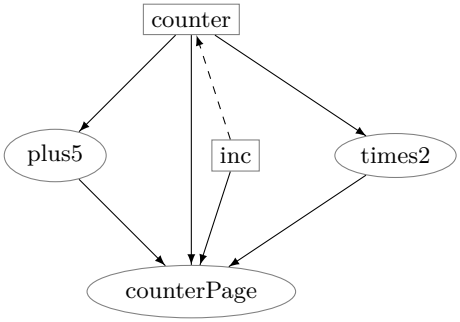

Figura 7. Reconfiguração do exemplo do contador.

```
def x = counter + 1
def y = x + 1
def x = y + 1
```

Figura 8. Dependência cíclica.

reconfigurado para incluir os valores de `plus5` e `times2`. A ação `inc` também é reconfigurada para incrementar o contador em 2 valores, ao invés de apenas 1. Por fim, o nome `plus5` é reconfigurado de **`var`** para **`def`**, de modo a ser recalculado sempre que o contador é incrementado.

O sistema de tipos garante, de forma estática, que a evolução das aplicações é feita de forma segura, sem causar disrupções ao sistema, ou seja, a introdução de incompatibilidade de tipos ou a inserção de ciclos no grafo de dependências (representado na Figura 7). Na Figura 8 está representado um exemplo em que a última instrução introduz um ciclo: os nomes `x` e `y` ficam a depender mutuamente um do outro. Isto faria com que a propagação de alterações entrasse num ciclo infinito, logo a instrução é rejeitada pelo sistema de tipos, não sendo introduzida no sistema. Um exemplo de uma mudança de tipo insegura seria a execução do seguinte comando: `var counter = "foobar"`. Uma vez que existem nomes que dependem de `counter` como número (inteiro), este não pode mudar de tipo para *string*. Portanto, neste caso, o sistema de tipos também rejeita este tipo de reconfigurações à aplicação.

3 Sistema de Suporte à Execução

O sistema de suporte à execução é constituído por 3 componentes principais:

- **Interpretador** – Interpreta, verifica estaticamente e executa o código das aplicações.
- **Base de dados** – Armazena persistentemente os dados e código das aplicações.

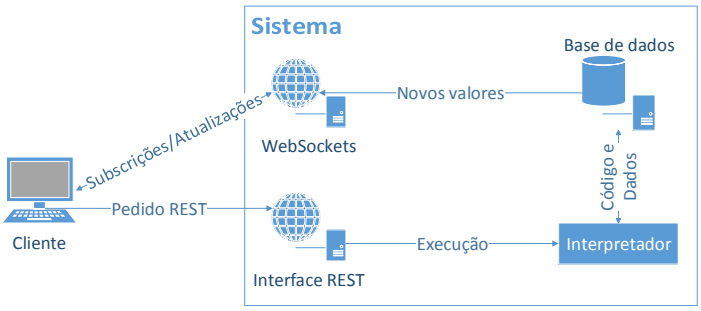

Figura 9. Arquitetura do sistema.

– **Servidor *web*** – Recebe pedidos dos clientes e envia atualizações via WebSockets.

O modo como estes componentes interagem entre si está representado na Figura 9. De seguida, descreve-se cada um dos componentes da arquitetura.

O interpretador é o componente central de todo o sistema, sendo responsável por implementar a linguagem apresentada na Secção 2. O interpretador executa o código de cada aplicação, verificando-o estaticamente através do sistema de tipos, garantido que cada operação permite evoluir a aplicação sem qualquer disrupção. Sempre que o interpretador altera o valor ou a expressão associada a um nome, o sistema utiliza o grafo de dependências para adicionar a uma fila todos os nomes que dependem do nome modificado. O sistema percorre a fila e recalcula os novos valores de cada um dos nomes. Quando a fila fica vazia, significa que as alterações foram propagadas a todo o sistema. Estas recomputações são processadas sequencialmente. Um melhoramento futuro passa por paralelizar as recomputações dos nomes que não têm dependentes em comum.

A base de dados (MongoDB[1]) é utilizada para armazenar, persistentemente, o código e dados das aplicações. Várias informações relativas a todos os nomes declarados na aplicação são armazenadas na base de dados, como a AST, o último valor calculado (evitando recomputações desnecessárias), o tipo da expressão (utilizada pelo sistema de tipos na verificação estática dos incrementos de código) e as dependências associadas a esse nome (de modo a propagar alterações pelo grafo de dependências). Tanto o código como os dados são guardados na mesma base de dados de igual forma, recorrendo à serialização das AST e dos resultados, de modo a serem armazenados como *strings*. Sempre que se pretende obter código ou dados da base de dados, é feita a desserialização dessas *strings*.

O servidor *web* é o responsável por receber os pedidos dos clientes e gerar páginas HTML. Este servidor está dividido em duas interfaces: REST e WebSockets. A interface REST é utilizada sempre que a comunicação é iniciada pelo cliente. O método `GET` é usado para obter o valor associado a um nome. Caso esse nome seja uma função, os argumentos podem ser passados no URL. Por exemplo, caso se pretenda chamar uma função, de nome `page`, que recebe dois

[1] `https://www.mongodb.org`

```
def counterPage =
    let step = 2 in
        <div>
            <h1>"Counter"</h1>
            <p>"Counter: " counter</p>
            "New value: " <input type="number" id="newCounter" value=counter/>
            <button doaction=(action { counter := #newCounter })>
                "Set counter"
            </button>
            <br/>
            <button doaction=(action { counter := counter + step })>
                "Increment"
            </button>
        </div>
```

Figura 10. Página HTML do contador com duas ações.

números, o endereço seria `page/1/2`, onde 1 é o primeiro argumento da função e 2 o segundo. O método `POST` é utilizado pelo programador para adicionar novo código à aplicação, sendo esse código passado no corpo do pedido. Por fim, o método `PUT` é utilizado para executar ações que, tipicamente, estão associadas a botões ou outros eventos. Para tal, é necessário incluir no corpo do pedido o identificador da ação, o ambiente em que deve ser executada e os *inputs* preenchidos pelo cliente que sejam usados pela ação. A interface WebSockets é usada para enviar ao cliente os novos valores associados a um nome. Para tal, utiliza-se o padrão do observador, onde o cliente inicialmente subscreve a um conjunto de nomes, dos quais pretende receber sempre os valores mais atuais. Sempre que um desses nomes muda, o servidor envia ao cliente uma notificação com o novo valor, atribuindo propriedades reativas às aplicações. Por exemplo, quando o utilizador carrega a página representada nas Figuras 2 e 3, o *browser* envia, por WebSockets, um pedido a subscrever ao nome `counterPage`. Quando o contador é incrementado, a página é recalculada para exibir o novo valor. Consequentemente, o servidor envia a todos os subscritores da página `counterPage` uma mensagem a indicar qual o novo valor do elemento HTML que contém o contador.

De seguida, na Secção 3.1 é explicado em detalhe como é efetuado o processo de *hoisting* de nomes, que permite que funções parcialmente avaliadas sejam utilizadas nas páginas HTML. Na Secção 3.2 aborda-se como as páginas HTML são mantidas em consonância com os dados presentes no servidor. Por último, na secção 3.3, é apresentado o IDE usado para o desenvolvimento de aplicações.

3.1 *Hoisting*

A linguagem base suporta funções de primeira ordem com avaliação parcial. Também as ações contêm *closures* que podem ter a sua execução adiada e que estão, tipicamente, associadas a botões e outros eventos do cliente. Quando o utilizador pretende executar uma ação, envia um pedido ao servidor indicando qual o nome dessa ação, logo cada ação deve estar associada a um nome único. Contudo, esta ação pode estar definida *inline* (ver Figura 10), não estando

```
var counterPage$1 = action { counter := #newCounter }

var counterPage$2 = action { counter := counter + step }

def counterPage =
    let step = 2 in
        <div>
            <h1>"Counter"</h1>
            <p>"Counter: " counter</p>
            "New value: " <input type="number" id="newCounter" value=counter/>
            <button doaction=(counterPage$1)>
                "Set counter"
            </button>
            <br/>
            <button doaction=(counterPage$2)>
                "Increment"
            </button>
        </div>
```

Figura 11. Programa da Figura 10 após o processo de *hoisting*.

associada a nenhum nome único. Para resolver este problema, é preciso garantir que todas as ações têm um nome único associado. Isto pode ser alcançado recorrendo à técnica padrão de *hoisting*, i.e., colocar ações definidas *inline* num nome *top-level* e substituir a sua utilização por um apontador. Esta técnica é, normalmente, utilizada pelos compiladores de linguagens funcionais.

De seguida, apresenta-se um exemplo de como a linguagem lida com o *hoisting* de ações. Na Figura 10 está uma nova reconfiguração do `counterPage`. Esta reconfiguração tem por objetivo apresentar dois botões: um para definir o valor do contador e outro para incrementá-lo. Salienta-se o facto de ambos os botões recorrerem a ações definidas *inline*, sendo que o botão de incremento também utiliza valores de escopo. Quando o processo de *hoisting* é aplicado, este programa é transformado para o que está representado na Figura 11. O código HTML correspondente tem o aspeto ilustrado na Figura 12. Quando o utilizador prime um dos botões, o *browser* envia um pedido ao servidor indicando qual das ações pretende executar, bastando para isso indicar o identificador da ação.

No caso da página da Figura 12, essa ação pode ser `counterPage$1` ou `counterPage$2`. Caso a ação necessite de valores de ambiente para ser executada, como é o caso da ação `counterPage$2`, esses serão incluídos no objeto `env`, como representado na Figura 12. Assim, quando o utilizador indica ao servidor a intenção de executar uma ação que depende do ambiente, os valores necessários para a sua execução são enviados no corpo do pedido. Caso a ação utilize valores de *input*, tal como a ação `counterPage$1`, os nomes que são necessários enviar são indicados no vetor `args`. Desta forma, o *browser* sabe quais são os *inputs* que é necessário enviar ao servidor sempre que pretende executar uma determinada ação. Quando o botão associado à ação `counterPage$1` é pressionado, o cliente envia

```
<div>
    <h1>Counter</h1>
    <p>Counter: 0</p>
    New value: <input type="number" id="newCounter" value="0"/>
    <button doaction={
        "action": "counterPage$1",
        "env": {},
        "args": ["newCounter"]
    }>
        Set counter
    </button>
    <br/>
    <button doaction={
        "action": "counterPage$2",
        "env": {"step": 2},
        "args": []
    }>
        Increment
    </button>
</div>
```

Figura 12. HTML gerado pelo programa 11.

```
def f x =
    let a = 1 in
        y => x + y + a
```

Figura 13. Função que recebe dois parâmetros.

```
var f$1 = x + y + a

var f$2 =
    let a = 1 in
        <y, f$1, [a = 1]>

def f = <x, f$2, []>
```

Figura 14. Função da Figura 13 após o processo de *hoisting*.

ao servidor um pedido contendo o identificador da ação e os *inputs* necessários (o valor introduzido pelo utilizador no *input* com id `newCounter`).

***Hoisting* de funções.** O processo de *hoisting* também é aplicado ao corpo das funções. Na linguagem, as funções são de primeira ordem, podendo ser retornadas como valores e executadas mais tarde. Para tal, é preciso aplicar o processo de *hoisting* no corpo das funções, de modo a que os resultados não contenham código. Um exemplo de *hoisting* de funções está representado nas Figuras 13 e 14, onde se declara uma função `f` com o parâmetro `x` que devolve uma função anónima com o parâmetro `y`. Realça-se o facto de a função anónima de parâmetro `y` utilizar um nome de escopo definido fora do seu corpo. As funções são convertidas para tuplos com três elementos: o nome do argumento, o nome que contém o corpo da função e o ambiente da função. O nome `f` passa a conter um tuplo que indica qual o nome do argumento (neste caso, `x`) , qual o nome que contém o corpo da função

```
var empty = true

def page =
    if empty then
        <strong>"empty"</strong>
    else
        <em>"full"</em>
```

Figura 15. Página HTML com dois *templates* possíveis.

(neste caso, `f$2`) e qual o ambiente em que foi declarado (neste caso, o ambiente é vazio) O nome `f$2` contém o corpo da função de parâmetro `x`, devolvendo um tuplo correspondente à segunda função. A principal diferença deste tuplo e do anterior é a presença do ambiente, contendo o valor associado ao nome `a`. O corpo da segunda função está associado ao nome `f$1`. Os tuplos são tratados como apontadores para código. Sempre que se depara com um destes tuplos, o interpretador adiciona uma nova entrada ao ambiente da função (o terceiro elemento do tuplo), associando o nome do parâmetro (o primeiro elemento do tuplo) ao valor passado na aplicação da função. De seguida, utiliza este ambiente na execução do código associado ao nome que contém o corpo da função (o segundo elemento do tuplo).

3.2 Refrescamento de páginas

Todas as páginas mantêm uma ligação por WebSockets com o servidor, de modo a receberem atualizações caso algum valor presente na página seja atualizado. O servidor mantém uma lista de subscrições, contendo informações sobre os utilizadores e em que páginas se encontram. A subscrição de uma página guarda as dependências associados ao seu cálculo. Sempre que uma dessas dependências é alterada, a página é recalculada. Estas dependências são calculadas com base no sistema de tipos da linguagem, com a única diferença de serem armazenadas na subscrição da página e não na base de dados utilizada pelo interpretador.

Para evitar que o servidor reenvie ao cliente toda a página sempre que esta é modificada, uma subscrição também guarda o último valor calculado. Assim, quando a página sofre uma modificação, basta comparar o novo valor com o antigo e enviar apenas as alterações. Na versão atual do protótipo, esta comparação é feita apenas nos elementos HTML de mais alto nível. Caso a página utilize parâmetros passados no URL, estes também são guardados na subscrição.

Por fim, a subscrição guarda um identificador único associado ao elemento base da página HTML. Este identificador serve apenas como um auxiliar para se saber quando é desnecessário utilizar o algoritmo de comparação, sendo preferível reenviar toda a página. A Figura 15 ilustra esta situação, onde página `page` pode tomar dois *templates* distintos. Cada um destes *templates* HTML está associado a um identificador único. Quando o valor do nome `empty` é `true`, o nome `page` contém a *string* `empty` a negrito. Quando o nome `empty` muda para `false`, o nome `page` fica associado a um *template* diferente, com a *string* `full` a itálico.

Figura 16. IDE disponível através do *browser*.

3.3 Ambiente de Desenvolvimento

O nosso servidor [9] fornece um ambiente de desenvolvimento (IDE) que permite ao programador desenvolver aplicações *web* diretamente no *browser*. O IDE, apresentado na Figura 16, divide-se em três painéis:

1. **Editor** – É neste painel onde o programador pode, efetivamente, desenvolver as aplicações *web*, escrevendo código que será interpretado pelo servidor.
2. **Painel de resultados** – Este painel exibe os resultados retornados pelas expressões submetidas ao servidor. Este painel e o editor atuam como um *read–eval–print loop* (REPL), permitindo ao programador rapidamente experimentar código.
3. **Painel de nomes** – O painel de nomes tem o propósito de mostrar ao programador o estado atual da aplicação. Para tal, contém a lista de nomes definidos e o respetivo código e/ou resultados (as variáveis de estado – `var` – apenas mostram o valor corrente). Cada nome presente no painel dispõe de um conjunto de botões que executam diferentes operações. Por exemplo, é possível executar uma ação ou abrir uma página HTML diretamente no editor (como demonstrado na Figura 16 para o elemento `page`).

No seu estado atual, o IDE permite alcançar o objetivo de aproximar o programador da aplicação final, no caso de aplicações de pequena dimensão. Contudo, para facilitar o desenvolvimento de aplicações de maior dimensão, várias modificações teriam de ser feitas. Algumas dessas modificações são, por exemplo, o suporte a um editor de texto mais sofisticado, permitir separar os nomes por diversos módulos, editor de CSS e JavaScript, etc. Estes melhoramentos fazem parte do trabalho futuro do projeto.

4 Trabalho Relacionado

Há várias ferramentas e linguagens que implementam as funcionalidades presentes descritas neste artigo, como a reconfiguração dinâmica, a propagação de alterações e a reatividade. Nesta secção apresentamos dois modelos utilizados para implementar tais funcionalidades: *dynamic software updates* e programação incremental de *data-flow*.

Dynamic Software Updates. Uma forma de efetuar a reconfiguração dinâmica de sistemas é através do uso de *dynamic software updates* (DSU) [7]. Os DSU permitem modificar código em execução através da aplicação de *patches*, sem a necessidade de interromper o sistema. Kitsune [5] e Ekiden [6] são dois exemplos de ferramentas que seguem o modelo de DSU. Estas ferramentas não eliminam as fases de compilação e de *deployment*. O *patch* a ser aplicado precisa de ser compilado utilizando um compilador modificado para o efeito. Também a aplicação do *patch* precisa de ser controlada pelo programador, ao adicionar à aplicação código de suporte a reconfigurações dinâmicas durante a sua execução. Por vezes, também é necessário código para lidar com o processo de evolução da aplicação, onde se define como os dados devem ser traduzidas de uma versão do código para a seguinte. Esta evolução nem sempre é verificada estaticamente pelo compilador, o que pode resultar na introdução de novos *bugs* durante o processo de reconfiguração. Todos estes aspetos são tratados implicitamente pelo nosso sistema, que assegura a evolução segura do código e dados.

***Programação Incremental de* Data-Flow.** A nossa linguagem segue o modelo de programação incremental de *data-flow*, permitindo construir uma aplicação em pequenos incrementos que são introduzidos no sistema em execução sem interrupções. As linguagens que se enquadram nesta categoria mantém um grafo de dependências entre os nomes definidos no programa, de modo a propagar as alterações para todos os nomes. O grafo pode ser utilizado para conceder propriedades reativas às aplicações. Esta técnica é utilizada por linguagens como o SuperGlue [10] ou o Circa [3] para implementar ambientes de programação *live*, onde a aplicação em execução está sempre sincronizada com o código presente no editor, fornecendo ao programador *feedback* imediato sobre o seu trabalho, tal como no nosso projeto. Uma particularidade do Circa é o suporte a dois modos diferentes: textual e visual. Estes modos estão constantemente sincronizados, sendo possível fazer modificações ao programa através do modo visual (utilizando técnicas *drag and drop*) que são refletidas no código. A principal diferença entre estes projeto e o nosso é o facto de não se focarem em aplicações *web*, mas sim programas com o intuito de correrem localmente.

O LightTable [4] é um IDE que também aplica conceitos de programação incremental e de *data-flow* a linguagens mais tradicionais, como JavaScript, Python e Clojure. O IDE mantém a aplicação a correr em sintonia com o código presente no editor, de modo similar ao SuperGlue, Circa ou ao nosso projeto. Uma diferença do LightTable é também apresentar o valor corrente de uma variável diretamente no editor de texto. Tal como no caso do Circa e do SuperGlue, o

LightTable não está idealizado para o desenvolvimento de aplicações *web*, o que não permite evoluir uma aplicação em produção sem quebras de serviço.

5 Trabalho Futuro

O projeto apresentado neste artigo ainda se encontra em desenvolvimento, havendo funcionalidades importantes ainda por implementar, que serão apresentadas e discutidas nesta secção.

Múltiplas Aplicações. Uma das funcionalidades por implementar é o suporte para várias aplicações. Atualmente, o sistema suporta apenas uma aplicação. Com esta funcionalidade, seria possível correr diferentes aplicações, de forma isolada, no mesmo sistema. Estas aplicações poderão ser criadas como clones de outras. Isto permite ao programador lidar com aplicações de forma similar aos *branches* existentes no sistema de controlo de versões Git. Por exemplo, o programador pode fazer clone de uma aplicação, testar modificações no clone e, posteriormente, fazer *merge* do clone com a aplicação original. Esta funcionalidade pode ser útil quando o programador pretende simular a existência de um servidor de desenvolvimento, de modo a testar alterações sem as tornar visíveis a todos os utilizadores.

Valores de Sessão. Na versão atual do sistema, o estado é partilhado entre todos os utilizadores da aplicação. Isto previne a aplicação de guardar dados distintos para cada cliente (por exemplo, o nome de utilizador). Com valores de sessão, seria possível ter várias versões de um mesmo nome, i.e., o nome toma um valor diferente para cada cliente, o que aumentaria o leque de possibilidades do sistema. Por exemplo, é possível criar aplicações que possuam sistemas de *login*, mostrando informações diferentes a utilizadores distintos.

Persistência. Outra funcionalidade que pretendemos oferecer é a possibilidade de mapear dados diretamente na base de dados. No sistema atual, tanto dados como código são guardados na mesma base de dados, através da serialização das AST e dos resultados. Com esta funcionalidade, seria possível interagir diretamente com a base de dados, sem recorrer à serialização de objetos. Isto permitiria que o programador criasse tabelas de mais alto nível e efetuasse pesquisas sobre os dados de um modo muito mais eficiente.

JavaScript e CSS. De momento, o sistema apenas permite incluir código JavaScript e estilos CSS nas aplicação por intermédio dos atributos dos elementos HTML. Para permitir o desenvolvimento de aplicações mais complexas, a inclusão JavaScript e CSS é uma necessidade. A solução preferencial seria estender a linguagem atual com operações que permitam definir o estilo e o comportamento do lado do cliente da aplicação.

6 Conclusão

Neste artigo apresentamos um sistema de desenvolvimento de aplicações *web* com base numa linguagem incremental e reativa [2]. Como exemplificado nas Secções 2 e 3, a linguagem tem suporte para a definição de toda uma aplicação *web*: lógica, esquema de dados e interface. Combinando elementos de linguagens de *back-end* e de *front-end*, o sistema permite criar, em poucas linhas, aplicações *web* completas (desde a lógica do servidor à do cliente) com propriedades reativas.

O ambiente de desenvolvimento permite ao programador estar, permanentemente, consciente de qual o comportamento da aplicação *web* num ambiente de produção, ao combinar programação incremental com reatividade. Com o *deployment* implícito da aplicação e a possibilidade de a experimentar à medida que é implementada, o ciclo de evolução é encurtado e o programador consegue receber informação importante sobre o comportamento da aplicação mais cedo no ciclo. Acreditamos que ao fornecer *feedback* constante e imediato ao programador sobre o seu trabalho conduz a uma maior produtividade. Ao executar as reconfigurações sem a necessidade de reiniciar ou interromper o sistema, evita-se quebras de serviço durante o processo de evolução da aplicação.

Ao implementar as tarefas descritas na secção 5, as potencialidades oferecidas pelo sistema crescem, ao permitir utilizar funcionalidades habitualmente presentes em *frameworks web*, como sessões e armazenamento em base de dados.

Referências

1. Chen, Y., Dunfield, J., Acar, U.A.: Type-directed automatic incrementalization. In: ACM SIGPLAN Notices. vol. 47 (2012)
2. Domingues, M., Seco, J.C.: Typeful updates on reactive live web programming. Tech. rep. (2013)
3. Fischer, A.: Introducing circa: A dataflow-based language for live coding. In: Live Programming (LIVE), 2013 1st International Workshop on. IEEE (2013)
4. Granger, C., Attorri, R.: Light table, `http://lighttable.com/`
5. Hayden, C.M., Smith, E.K., Denchev, M., Hicks, M., Foster, J.S.: Kitsune: Efficient, general-purpose dynamic software updating for c. In: ACM SIGPLAN Notices. vol. 47 (2012)
6. Hayden, C.M., Smith, E.K., Hicks, M., Foster, J.S.: State transfer for clear and efficient runtime updates. In: 2011 IEEE 27th ICDEW (2011)
7. Hicks, M., Moore, J.T., Nettles, S.: Dynamic software updating. In: Proceedings of the ACM SIGPLAN 2001 Conference on PLDI (2001)
8. Lin, D.Y., Neamtiu, I.: Collateral evolution of applications and databases. In: Proceedings of the IWPSE-EVOL. ACM (2009)
9. Mateus, J., Domingues, M., Seco, J.C., Lopes, T., Martins, N.: Live programming prototype, `https://live-programming.herokuapp.com/`
10. McDirmid, S.: Living it up with a live programming language. In: ACM SIGPLAN Notices. vol. 42 (2007)