Optimization of the model-view-update pattern with partial evaluation

Liv Hartoft Borre and Jan Schill

IT University of Copenhagen, Copenhagen, Denmark
{livb,schi}@itu.dk

Abstract. Modern techniques to perform DOM updates on rendered HTML documents mostly use a VDOM with comparison of two views: the current view and an updated one after an action is performed. The comparison of tree-structures with subsequent patching of the structure and rerendering is computational expensive. With clever compiler optimizations using partial evaluation these comparisons can be reduced. An action should always make changes in a predictable manner and in the same position in the DOM. With partial evaluation the expensive comparison can be precomputed at compile time and reduced to a single direct change that circumvents the update-model-view cycle. This paper will cover this compile time optimization and the generation of JavaScript snippets, with which the change can be performed and demonstrate that the VDOM comparison can be prevented to a certain degree. The paper will not introduce the compiler to JavaScript that would generate a complete working program for the browser.

Introduction

Many ideas have been shaped in order to make websites more dynamic and avoid breaking the site when loading a new Hypertext Markup Language (HTML) file from the server. One pattern that solves this problem efficiently is the Model-View-Update (MVU) application pattern. To solve the problem of showing diverged information in an HTML document, the JavaScript language is needed, as there is no native and complex enough support for HTML or Cascading Style Sheets (CSS) to force the browser to show this new information. JavaScript on the other hand can directly inject into the Document Object Model (DOM), by having complete access to every HTML node element in the rendered DOM. JavaScript can select a node element by `ID` and change its displayed value, forcing the browser to rerender the DOM on that node. Figure 1 shows a minimal example, where JavaScript is used to alter the rendered HTML.

This way of updating content on the document directly has been evolved over the years and is heavily used to build any sized modern websites. With the help of frameworks like React[1], a JavaScript framework developed by Facebook

[1] `https://reactjs.org/`

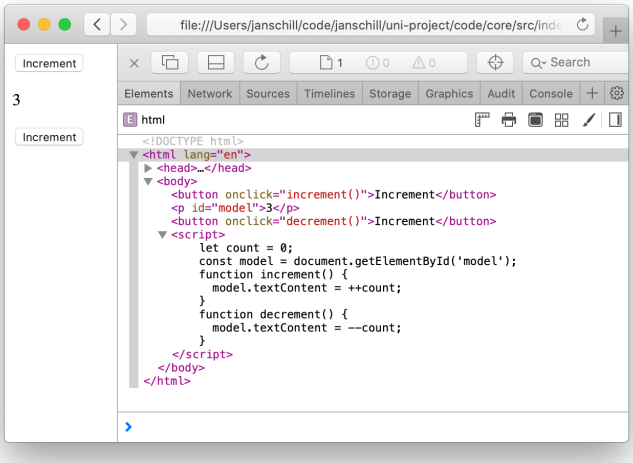

Fig. 1. Simple button counter example

or Elm[2], a framework utilizing its own functional language, it has become easier to go from a small button example to a fully working componentized website. The Elm framework uses the MVU application pattern to realize the idea of dynamically updating the DOM.

The basic functionality how Elm uses it, is as follows. The framework generates HTML that can be viewed by a client. This client can then interact with the HTML by clicking a button for example. This produces a `Msg (Message)`, which is send to Elm, that generates another HTML document. This cycle is what is referred to as the MVU cycle. In the Elm program three parts play a crucial role: The Model, View and Update.

When a message of type `Msg` is produced, the update function is being called, which takes the message and the current state of the model and returns an updated model. The view function, which is responsible of generating the HTML that can be viewed by the browser, is then called automatically, rendering the HTML with the updated model.

Listing 1.1. View, update and model and their types

```
1 view : Model -> Html Msg
2 update : Msg -> Model -> Model
```

How does Elm optimize the view rendering? Oftentimes when a new view is generated and needs to be re-rendered, it is not necessary for the whole document to be switched out, but rather only the DOM node that experienced

[2] `https://elm-lang.org/`

change. Forcing the browser to generate new DOM nodes is computational expensive work. For this most frameworks use a Virtual Document Object Model (VDOM), which resembles the complete active DOM in memory in a convenient data structure. The VDOM is generated with the updated model and compared against the current active VDOM, finding the parts that need updating. This part is called *diffing* and it is done by iterating over both representations of the DOM and comparing each individual node. This *diffing* then generates a data structure that keeps track of all changes. A *patch* function takes the old view and the data structure holding the changes, which then updates all occurrences of change in the view.

All the *diffing* and *patching* against the DOM tree seems to be still computational expensive, as in the end, only the following code snippet is actually needed for the button example to work:

Listing 1.2. Reduced example to show only the incrementing

```
<html>
  <button onclick=increment()>Increment</button>
  <p id="model">0</p>
</html>
<script>
  let count = 0;
  const model = document.getElementById('model');
  function increment() {
    model.textContent = ++count;
  }
</script>
```

How does JaLi optimize the view rendering even further? This paper will outline all the needed parts to use partial evaluation, symbolic execution and constant folding to optimize the view rendering. It will be shown on the small button example how the compiler of the JaLi language can reduce all the previously mentioned parts that are computational expensive, by applying the optimizations. Due to time restrictions and the sheer complexity of the project it was not able to implement the full working compiler optimizations, but enough to demonstrate reduction on certain programs. A compiler that transforms a full program to a HTML and JavaScript website that can run in the browser is also not implemented—this will be simulated by manual coding the JavaScript parts. Nevertheless, the theory has been established and it will be shown on small examples to illustrate the idea.

In the following chapter the functional language that was developed for this project will be explained. The next chapter describes partial evaluation and how reduction is carried out on the JaLi language. Next the implementation of the MVU in the JaLi language is shown, and lastly the button example will be used to show how it can reduced into functioning HTML and JavaScript that avoids the MVU cycle and the VDOM diffing.

1 The JaLi language

The JaLi language is a minimal higher-order functional language without type checking. The syntax is closely coupled to FSharp and Haskell.

Listing 1.3. Button example written in JaLi

```
1  type Node = Text String | Tag String [Tuple] [Node];
2  type Msg =
3      Increment
4    | Decrement;
5
6  model = 0;
7
8  func update msg val =
9    match msg with
10     | Increment -> val + 1
11     | Decrement -> val - 1
12 end
13
14 func view val =
15   Tag ('div') [('class', 'body')] [
16     Tag ('button') [('onClick', update (Increment) (
           val) )] [Text ('Increment')],
17     Tag ('p') [] [Text (val)],
18     Tag ('button') [('onClick', update (Decrement) (
           val) )] [Text ('Decrement')],
19   ]
20 end
21
22 view model
```

Listing 1.3 shows how the button example from the previous chapter could look like in the JaLi language. It introduces an algebraic data type (ADT) called `Node` in line 1, that has two constructors: `Text` and `Tag`. An ADT is a composition type, which means that it is used to introduce new and more complex types to a language. This is especially useful when wanting to depict a type from the outside into the language. This is illustrated by the `Node` type in the given example. The `Node` type is representing an HTML document element. In HTML an element always carries an element name, for example `button`: `<button></button>`. It has a list of key-value pairs, named as `attributes`: `id=buttonIncrement` , which are set after the element name in the opening tag. Lastly, it holds elements or text in between its opening and closing tags.

Listing 1.4. Button increment in HTML

```
1 <button id=buttonIncrement>Increment</button>
```

Represented in JaLi as an ADT the single button HTML element would look like in listing 1.5.

Listing 1.5. Button increment in JaLi as ADT

```
1 Tag ('button') ([('id', 'buttonIncrement')]) ([
2   Text 'Increment'
3 ])
```

Just like in HTML then, the ADT can recursively be nested to construct a tree-like structure. Another example for this would be a recursive list data type

Listing 1.6. Recursive list data type

```
1 type List = Nil | Cons Integer List;
2
3 Cons 1 (Cons 2 (Cons 3 (Nil)))
```

An ADT always carries super-type defined right after the keyword `type`, followed by a number of constructors, which are used to instantiate an actual value of the defined ADT. These constructors can carry optional type parameters, which need to be given, when creating it. Like in listings 1.4 and 1.5 the `Tag` constructor needs a string and two lists. It should be noted that the JaLi language it untyped, and that these types carry nothing but descriptive values as well as define the arity of the constructor, i.e. the number of arguments it requires.

ADTs make mostly only sense with pattern matching implemented as well. Pattern matching will allow to control the flow of the program according to the state of the matching ADT values.

Listing 1.7. Pattern matching on Msg

```
1 type Msg = Increment | Decrement;
2 message = Increment;
3
4 match message with
5   | Increment -> 'Increment the model'
6   | Decrement -> 'Decrement the model'
7   | _ -> 'Unknown action'
```

The example in listing 1.7 shows the known ADT definition in line 1. After it on line 2 a *let binding* can be seen, where the value on the right side is assigned to the variable called `message`. Afterwards this binding is used in pattern matching context. The value of `message` will be matched with the available patterns defined after the pipe symbol. There are three patterns that can be matched with, each valid action and one *wild card* pattern, that acts like the *else* block in an *if statement*. The complete `match` block will then return only the right side of the matched pattern.

Functions are defined by enclosing a name, parameters and a function body between the key words `func` and `end`.

Listing 1.8. Inner functions and recursion shown on factorial

```
1  func factorial n =
2    func mult x y =
3      x * y
4    end
5
6    if n == 0
7    then 1
8    else mult (n) (factorial (n - 1))
9  end
10 factorial 5 // => 120
```

Functions can be recursively called and can even have inner functions.

1.1 Grammar

The lexer, parser and interpreter are all written in FSharp using the libraries *FsLex* and *FsYacc*.

Lexer The lexer receives a JaLi program as a string, recognizes and transforms its characters in the program to tokens in FSharp. These tokens are defined in the *Lexer.fsl* and *Parser.fsy*. The lexer does this by having a defined list of regular expressions to tokens and matches the input string character by character to those patterns. The result is a list of tokens that might end up looking lik listing 1.9

Listing 1.9. Lexer result

```
1 // JaLi program as string:
2 "5 + 10"
3 // Result of lexing the string:
4 Parser.token = CONSTINT 5
5 Parser.token = PLUS
6 Parser.token = CONSTINT 10
7 Parser.token = EOF
```

After the whole string is successfully transformed into tokens, the parser will try to make sense of it.

Parser The parser receives the list of tokens after the lexical analysis and builds an abstract syntax tree (AST) from it. It does this by recognizing different combinations of tokens, that are defined explicitly as showcased by listing 1.10.

In the case of JaLi, a program—indicated by is point of entry with `Main`—is an `Expression`, which is defined in the parser as the following:

Listing 1.10. Program defined by `Expression`

```
1  Main:
2      Expression EOF { $1 }
3
4  Expression:
5    | AtomicExpression { $1 }
6    | FunctionCall { $1 }
7    | ConditionalExpression { $1 }
8    | Baseoperation { $1 }
9    | Binding { $1 }
```

From this, it is not obvious how a program then can be multiple `Bindings` or `Functions` with a `FunctionCall` in the end. This is realized by defining a `Binding` by its `Expression` that it binds, but also expecting another `Expression` right after the binding.

Listing 1.11. Binding with `Expression` afterwards

```
1  Binding:
2    | LocalBinding { $1 }
3    | FunctionBinding { $1 }
4    | ADTBinding { $1 }
5
6  LocalBinding:
7    | NAME ASSIGN Expression SEMICOLON Expression { Let($1, $3,
         $5) }
```

The `SEMICOLON` marks an end of a `Binding`, indicating that the `Expression` after it, is the next `Expression` in the program—allowing multi-line programs and not just a single line program. The part in curly braces after a parser rule was up until here completely ignored but shall be explained as being the mapping to its AST element. A `Binding` in the AST is represented by a `Let` key word, which holds three values, these are passed over by the call on line 7 in listing 1.11. The `Let` is specified as `Let of string * Expr * Expr` in the AST file. This means that the first parameter needs to be of type string, the name and the other two parameters of type `Expr`, which is a defined type.

Listing 1.12. An excerpt of the AST of JaLi

```
1  type Value =
2      | IntegerValue of int
3      | BooleanValue of bool
4      | CharValue of char
5      | StringValue of string
6      | TupleValue of Value * Value
7      | ListValue of Value list
8      | ADTValue of string * string * Value list
9      | Closure of string * string list * Expr * Value Env
```

```
10      | ADTClosure of ADTConstructor * string * Value Env
11
12  and Expr =
13      | ConcatC of Expr * Expr
14      | List of Expr list
15      | Constant of Value
16      | StringLiteral of string
17      | Variable of string
18      | Tuple of Expr * Expr
19      | Prim of string * Expr * Expr
20      | Let of string * Expr * Expr
21      | If of Expr * Expr * Expr
22      | Function of string * string list * Expr * Expr
23      | ADT of string * ADTConstructor list * Expr
24      | Apply of Expr * Expr list
25      | Pattern of Expr * (Expr * Expr) list
```

The listing 1.12 gives a rough overview of the multiple `Expressions` that are currently implemented and reveal a bit of its possibilities.

Returning back to the grammar in the parser the implementation and characteristic of the JaLi grammar shall be explained. As already mentioned, everything reduces to an `Expression`, which are split into the ones shown in listing 1.10.

We distinguish between `Atomic Expression` and other expressions. An `Atomic Expression` is a constant, a variable, a tuple, a list, a cons operator, or a parenthesized expression (so that it is easy to see where an `Atomic Expression` begins and ends). Constants are integers, booleans, strings, and the wildcard character '_' which is used for match expressions.

The reason we make this distinction is because syntax like function applications without parentheses makes the language ambiguous, as we then cannot decide when the function application expression ends, and another begins. A solution to this, is to distinguish between `Atomic Expression` and other expressions, and then require arguments for function application to be atomic.

Besides `Atomic Expression`, an expression can be a logical negation, a base operation for arithmetic and comparison (+, -, ==), a conditional expressions, a match expressions, a function call or a binding. A binding is an ADT declaration, a local binding, or a function declaration. Match expressions consist of a match expression and a list of patterns. Each pattern is a tuple of expressions, where the first expression is the pattern to match against the match expression, and the second is the body to execute if the pattern matches. The ADT consist of a name and a list of constructors. Each of these constructors again contain a name and the list of type parameters. The types describe the arity of the constructor, and provides descriptive value, but it does not imply any actual type constraints on its arguments.

```
Main ::=
  expr

expr ::=
  atomicexpr                              atomic expression
  !expr                                   logical negation
  expr op expr                            base operations, arithmetic,
                                          comparison
  if expr then expr else expr             conditional expression
  match expr with patterns                match expression
  NAME atomicexprs                        function call
  binding                                 types, locals, functions

atomicexpr ::=
  const                                   constant literals
  NAME                                    local variable or parameter
  (expr, expr)                            tuple
  [ expr, expr, ..., expr ]               list
  expr::expr                              cons operator
  ( expr )                                parenthesized expression

const ::=
  CONSTINT                                integer literal
  CONSTBOOL                               boolean literal
  STRINGLITERAL                           string literal
  USCORE                                  wildcard literal

binding ::=
  type NAME = constructors; expr          abstract data type binding
  NAME = expr; expr                       local binding
  function NAME params = expr end expr    function binding

patterns ::=
  | expr -> expr                          one pattern
  | expr -> expr patterns                 more than one pattern

constructors ::=
  NAME typeparams                         one type constructor
  NAME typeparams | constructors          more than one type constructor

type ::=
  INT                                     Integer type
  FLOAT                                   Float type
  BOOLEAN                                 Boolean type
  STRING                                  String type
```

```
46    CHAR                                  Char type
47    NAME                                  Name variable type
48    ( type, type )                        Tuple of types
49    [ type ]                              List of type
50
51  typeparams ::=
52    (* empty *)                           zero type parameters
53    type typeparams                       more than zero type parameters
54
55  atomicexprs ::=
56    atomicexpr                            one expression
57    atomicexpr atomicexprs                more than one expression
58
59  params ::=
60    NAME                                  one parameter
61    NAME params
```

Interpreter An interpreter takes an expression, evaluates it and returns a value. The interpreter implemented for JaLi is also written in FSharp. When given an expression from the list defined in the AST, it matches this expression with the correct pattern and returns its value.

Listing 1.13. Function head of `eval` function in interpreter

```
let rec eval (e: Expr) (env: Value Env): Value =
    match e with
    // ...
    | And (expression1, expression2) ->
        match eval expression1 env with
        | BooleanValue false -> BooleanValue false
        | BooleanValue true -> eval expression2 env
        | _ -> failwith Can only use boolean values
    // ...
```

Listing 1.13 shows the head of the `eval` function. It takes two arguments: The expression and an environment variable of type `Env`, which is a list of tuples of type string and the type parameter provided. This environment is used to store variables and functions. This will be explained further on a concise example. The pattern matching from listing 1.13 gives the evaluation of an `And` expression, which is the logical conjunction. It operates on two expressions, but because an expression is the base case for the JaLi language, those expressions can of course only be evaluated to a boolean value, if they are boolean values themselves. Therefore, both expressions need to be evaluated to the type `BooleanValue` before they can be operated on. In the logical conjunction it is only necessary to evaluate the second expression, when the first evaluates to true, because if the first expression is false, it does not matter what the second expression holds, as it will never evaluate to true.

The interpreter also needs to account for the problem of wanting to have more than just one single expression in a program. Those were handled for example at *let bindings*, which consists of two expressions, the one it binds to and all expressions after it.

Listing 1.14. Evaluation of let bindings

```
1     // ...
2     | Let (name, expression1, expression2) ->
3         let value = eval expression1 env
4         let newEnv = (name, value) :: env
5         eval expression2 newEnv
6     // ...
```

Because a *let binding* is essentially not a full program by itself, as it only binds an evaluated expression to a variable and not returning it by itself until it is explicitly called, it needs to do three operations:

1. It needs to evaluate the expression that it is supposed to bind to.
2. It needs to use the provided environment, to make this variable and its expression available for expressions following it.
3. It needs to continue the interpretation of the program by evaluating the second expression, while providing the new environment

One interesting aspect about evaluating programs are the handling of functions. Function declarations will be—as previously explained—put into the environment, before this happens though, they are wrapped in a type `Closure`, which holds the function name, the parameters it expects, the current environment and the expression, which is the function body.

Listing 1.15. Evaluation of function bindings

```
1     // ...
2     | Function (name, parameters, expression, expression2) ->
3         let closure =
4             Closure(name, parameters, expression, env)
5         let newEnv = (name, closure) :: env
6         eval expression2 newEnv
7     // ...
```

When the AST type `Apply`—the call of a function—is encountered, it first looks up the `Closure` in the environment, then makes sure that not too many arguments are provided, if less arguments are given then the function is defined with, it will return another `Closure`, partially applying the function. This will be important for partial evaluation in chapter 2 to make the optimizations that are planned. In both cases of less arguments and equal arguments to the parameters, the interpreter will bind those values, to the parameters, making them available for the function body to apply.

Listing 1.16. Binding of arguments and parameters in a function call

```
let newEnv =
    List.fold2 (fun dEnv parameterName argument ->
      (parameterName, eval argument env) :: dEnv)
      ((cname, fclosure) :: declarationEnv) cparameters
          farguments
```

Another important and interesting case is the pattern matching evaluation. For this a `Pattern` type with the `matchExpression` and a list of patterns is mapped on. Firstly, the `matchExpression`, which is the expression that is desired to find in the patterns, is evaluated. The value from this operation is then being matched against the different patterns given and in general available.

Listing 1.17. Evaluation of pattern matching

```
    // ...
    | Pattern (matchExpression, (patternList)) ->
        let matchPattern x (case, expr) =
            tryMatch (tryLookup env) x case
            |> Option.map (fun bs -> (case, expr, bs))

        let evaluatedMatch = eval matchExpression env
        match List.tryPick (matchPattern evaluatedMatch)
            patternList with
        | Some (case, expr, bindings) -> env @ bindings |> eval
            expr
        | None -> failwith Pattern match incomplete
    // ...
```

Listing 1.18. Helper function to find pattern and bind values

```
let rec tryMatch (lookupValue: string -> option<Value>)
                 (actual: Value) (pattern: Expr) =
    let tryMatch = tryMatch lookupValue

    match (actual, pattern) with
    | (_, Constant (CharValue '_')) -> Some []
    | (a, Constant v) when a = v -> Some []
    | (a, Variable x) ->
        match lookupValue x with
        | Some (ADTValue (a, b, c)) -> tryMatch actual (Constant
            (ADTValue(a, b, c)))
        | _ -> Some [ (x, a) ]
    | (ADTValue (name, _, values), Apply (Variable (callName),
        exprs)) when name = callName ->
        forAll tryMatch values exprs
    | (TupleValue (v1, v2), Tuple (p1, p2)) ->
```

```
15          match (tryMatch v1 p1, tryMatch v2 p2) with
16          | (Some (v1), Some (v2)) -> Some(v1 @ v2)
17          | _ -> None
18      | (ListValue (valList), List (exprList)) when valList.
            Length = exprList.Length -> forAll tryMatch valList
            exprList
19      | (ListValue (h :: t), ConcatC (h', t')) -> forAll tryMatch
              [ h; (ListValue t) ] [ h'; t' ]
20      | _, _ -> None
```

The variable `actual` is the value from the `matchExpression` and `pattern` the current pattern from the list of patterns, passed in shown on line 4 on listing 1.17, with the `patternList` from line 8. Some interesting cases in the `tryMatch` are for example the *wildcard* pattern: `(_, Constant (CharValue'_'))`, this means that if the pattern is an underscore symbol, it should be matched no matter the actual value of the `matchExpression`. It is worth noting that the `tryMatch` function returns a *Some* with a list of tuples. The tuples are bindings of the pattern to the match value, this makes the value available for further use in the body of the pattern match. When all bindings are collected the correct match-pattern combination is being searched for. If found, it will be added to the environment and the body of the pattern evaluated.

2 Reducer

The essence of what we are going to do is that we are going to exploit as much of the statically known data in the program, by reducing it into a smaller residual program, which is then compiled into HTML and JavaScript. The program we are reducing is the MVU described in the introduction 1. Reducing this program will allow us to optimize the entire diffing cycle, because we can generate JavaScript functions that know exactly what document elements to change, when called. This is all to achieve improved efficiency compared to current VDOM diffing. However, in order to understand and apply it, partial evaluation needs to be understood.

2.1 Partial evaluation

When all inputs to a program are given, an interpreter evaluates the program, and obtains a result. However, this is possible only when all of the inputs are known. Partial evaluation is a program optimization technique concerned with specialization and evaluation of programs where only parts of the inputs are known.

Most developers are familiar with specialization from partial application of functions. Partially applying a two-argument function obtains a one argument function where the first value has been *fixed* to the given value. Fixing the variable to a specific value is called specialization.

A partial evaluator is an algorithm which, when given a program and some of its input, will attempt to execute the program as far as possible, and output a reduced *residual* program. When given the remaining inputs the residual program will execute the rest of the program. It works as if the input has been *incorporated* into the original program, as much as possible has been evaluated, and unnecessary branches have been reduced away. Evaluation of the residual program with the remainder of the inputs should yield the same output as evaluation of the original program with all of the inputs. In that sense, it is a specialized version of the original program. Partial evaluation is also known as *program specialization*.

The below Figure 1.19 is an example from [1] showing a two-input program p for computing x^n. Partially evaluating the program with the static parameter 5 yields the second specialized program p5.

Listing 1.19. Specialization of a program to compute x^n

```
1  f(n,x) =
2    if n = 0 then 1
3    else if even(n) then f(n/2,x)^2
4    else x * f(n-1,x)
5
6  f5(x) = x * ((x^2)^2)
```

The reason why this is beneficial is for efficiency gains. Partially evaluating the program with 5 yields the program `p5` where this expensive computation has already been done. Thus `p5(x)` is a much faster program than `p(5)(x)`. It is worth to note that this optimization is possible because n determines the control of the program. If `x` was the static parameter, it would not be possible to achieve the same optimization.

Figure 1.20 shows another example, of the two-input function `update`, which either increments or decrements the model by one, based on the given `msg`. Partially evaluating the function with the static parameter `Increment` yields a new function which always adds 1 to the model it receives.

Listing 1.20. Specialization of the update function on increment

```
1 func update msg model =
2   match msg with
3   | Increment -> model + 1
4   | Decrement -> model - 1
5 end
6
7 func updateIncrement model =
8   model + 1
9 end
```

Let's assume that each branch contains some expensive computation. Partially evaluating `update` with `Increment` yields the function `updateIncrement` where this expensive computation has already been done. This is desirable when the function is called with several different second parameters, as it allows to reuse `updateIncrement` with several different arguments without recomputing the expensive computation. Thus efficiency can be achieved by partially evaluating the `update` function with each of the inputs `Decrement` and `Increment`, and replace all calls to `update(Increment)` with the specialized function `updateIncrement`, and replace all calls to `update(Decrement)` with the reduced function `updateDecrement`.

2.2 Approach

So what is actually going on in partial evaluation, is a combination of evaluation and code generation. We are evaluating all calculations that depend only on the known input. These expressions are called *static*. All expressions that rely on unknown inputs are called *dynamic*. For each dynamic expression we generate code by replacing it with a new expression. This is described by the following techniques. There are three main techniques in partial evaluation [1]: *symbolic computation*, *unfolding function calls*, and *program point specialization*. The two first techniques have been sufficient for this project and will be described here, while the third one will be described in improvements.

Symbolic computation: This is the process of computing with symbolic values either by rewriting or evaluation. Symbols, in this context expression, represent rewritable terms, while values imply the end of rewritability. We are going to specialize function bodies by reducing it symbolically; we will rewrite dynamic expressions to other expressions (code generation) and evaluate static expression into values.

Unfolding: This means replacing a function call by a copy of `f's` body where all parameter variables have been replaced by the corresponding arguments. E.g. if there is somewhere in the program that the partial evaluator finds a call to `unfold Increment 5` it will replace the call with the constant 6.

Unfolding Strategy: Unfolding can either be done *on the fly* or in a post phase. We will be using the former. To avoid infinite unfolding, Sestoft [1] proposes three strategies.

1. Avoid infinite unfolding by not unfolding function calls, however this would mean that the function calls would not be unfolded even though the arguments are static, resulting in a minimal specialization.
2. Only unfold function calls when all parameters are static. This risks only infinite unfolding if the original program had a potential *infinite static loop*, i.e. a loop that does not involve any dynamic tests.
3. Not to unfold a function call inside a dynamic conditional. As before, this avoids infinite unfolding as long as the original program does not contain a potential infinite static loop itself.

We will be using the latter strategy, however since the control flow in our language is also determined by pattern matching, we will extend the constraint to also apply in dynamic match expressions.

Offline and online partial evaluation: One should note that there are two different processes for partial evaluation: online and offline. Offline divides the partial evaluation into two stages. The first is a preprocessing phase, where expressions are divided in static and dynamic expressions, e.g. by annotating each expression as either dynamic or static, based on the available information. Then during the actual specialization, the decision of whether a certain expression should be reduced or not is based on the precomputed division. Online processing consists of only one phase: the action to take at each expression is decided based on the concrete values computed during specialization. The main advantage of the online process is that it exploits more static information during specialization than the offline process [1]. The online process will be followed.

2.3 Implementation

Our partial evaluator is going to produce a residual program by running through the program and pre-compute as much as possible, based on static variables and

arguments to function calls. Having detailed the expressions of the JaLi language in Chapter 3, here it will be explain how each of them are reduced by the partial evalutor.

A note on the implementation: The reduction of the expressions is mostly trivial, however reducing the patterns are quite complex. We believe that this is mainly caused by the abstract syntax behind JaLi, mainly in the different representations of ADTs. A much simpler implementation of the reducer is very likely, with a better implementation of the abstract syntax. Another notable thing is that this is not an optimal implementation of the reducer, as it may reduce function bodies and other expressions even though no static information is available and thus no reduction will be achieved. For now, this works just fine for the examples demonstrated.

The partial evaluator is implemented as the single function `reduce` as seen in listing 1.21. As main parameters it takes an expression to reduce and a store. The store is a list of variable bindings similar to the environment parameter in the interpreter; it is a list of (string, Expr) tuples, mapping a name to an expression. The last parameter is the *context* which expresses whether it is inside a dynamic conditional. This is to implement the unfolding strategy that does not unfold function calls inside dynamic conditionals. When done, the reduce function outputs an Expr.

Reducing a constant returns the constant. Reducing a variable will look up the variable in the store and return the expression.

Listing 1.21. The reduce function

```
let rec reduce
    (e: Expr)
    (context: bool)
    (store: Expr Env): Expr =
    match e with
    | Constant c -> Constant c
    | Variable x -> lookup store x
```

Listing 1.22 shows the reduction of concatenations, tuples and lists, as these cases resemble each other. We reduce each of the sub-expressions. For concatenation and lists we return a list and for tuples we return a tuple with the reduced sub-expressions.

Listing 1.22. Reducing concatenations tuples and lists operations

```
| ConcatC (h, t) ->
    let head = reduce h context store
    let tail = reduce t context store
    match (head, tail) with
    | (v, List (vs)) -> List(v :: vs)
```

```
6       | _ -> ConcatC(head, tail)
7   | Tuple (expr1, expr2) ->
8       let r1 = reduce expr1 context store
9       let r2 = reduce expr2 context store
10      Tuple(rexpr1, rexpr2)
11  | List (list) ->
12      let reducedItems =
13          List.map (fun e -> reduce e context store) list
14
15      List(reducedItems)
```

As shown in listing 1.23, a primitive operation is reduced similarly to the previous, by reducing each sub-expression. If both are constant, the interpreter will be used to evaluate the primitive expression and return a constant value. However, even though one of the sub-expressions are not constant, it may still be possible to reduce the expression. E.g. multiplying a variable with 0 will always yield zero, as seen in the match cases in the listing. For the problems that are being showcased, it suffices to consider only these cases. However, the list is not exhaustive, and a complete partial evaluator should consider many more cases in order to be optimal.

Listing 1.23. Reducing primitive operations

```
1   | Prim (op, expr1, expr2) ->
2       let rexpr1 = reduce expr1 context store
3       let rexpr2 = reduce expr2 context store
4       match op, rexpr1, rexpr2 with
5       | _, Constant _, Constant _ -> Constant <| eval (Prim(op,
            rexpr1, rexpr2)) []
6       | "*", _, Constant (IntegerValue 0) -> Constant <|
            IntegerValue 0
7       | "*", _, Constant (IntegerValue 1) -> rexpr1
8       | "+", _, Constant (IntegerValue 0) -> rexpr1
9       | "-", _, Constant (IntegerValue 0) -> rexpr1
10      | "*", Constant (IntegerValue 0), _ -> Constant <|
            IntegerValue 0
11      | "*", Constant (IntegerValue 1), _ -> rexpr2
12      | "+", Constant (IntegerValue 0), _ -> rexpr2
13      | "-", Constant (IntegerValue 0), _ -> rexpr2
14      | "==", Variable x, Prim("+", Variable x1, _) when x = x1
15              -> Constant <| BooleanValue false
16      | "==", Variable x, Prim("+", _, Variable x1) when x = x1
17              -> Constant <| BooleanValue false
18      | "==", Variable x, Prim("-", Variable x1, _) when x = x1
19              -> Constant <| BooleanValue false
20      | "==", Variable x, Prim("-", _, Variable x1) when x = x1
21              -> Constant <| BooleanValue false
```

```
22      | _ -> Prim(op, rexpr1, rexpr2)
```

Listing 1.24 shows reduction of let-bindings. When reducing let-expressions the body will be reduced first. Then the result will be added to the store, bound to the name of the binding variable. Lastly the following expression will be reduced. This effectively removes the let binding expressions. The expression lives in the store and replaces dynamic variables at the places where it is referenced.

Listing 1.24. Reducing let bindings

```
1  | Let (name, expr1, expr2) ->
2      let e = reduce expr1 context store
3      (name, e) :: store |> reduce expr2 context
```

Reducing conditionals is shown in listing 1.25. First, the conditional expression is reduced. If the expression is a constant boolean value, we already know what branch the program will take, and thus we return the reduction of this branch. If the conditional is not a constant, we return a conditional expression with the branches reduced. As according to our unfolding strategy, we change the context to false, expressing that we are inside a dynamic conditional, and thus should not unfold function calls when encountering them.

Listing 1.25. Reducing conditional expressions

```
1  | If (cond, thenExpr, elseExpr) ->
2      let rcond = reduce (cond) context store
3      match rcond with
4      | Constant (BooleanValue (b)) ->
5          if b
6          then reduce (thenExpr) context store
7          else reduce (elseExpr) context store
8      | _ -> If(rcond,
9              reduce (thenExpr) false store,
10             reduce (elseExpr) false store)
```

Listing 1.26 shows the reduction of functions. In order to handle recursive calls, we add the closure to the store before reducing the function body. Then we create a new closure with the reduced body, add it to the original store, and continue by reducing the second expression that follows the function. Here we always return the reduced expression that follows the function. The function exists as a closure in the store, which will be specialized and unfolded at function calls.

Listing 1.26. Reducing functions

```
1  | Function (name, parameters, body, expr2) ->
2      let bodyStore =
3          (name, Constant <| Closure(name, parameters, body, []))
4          :: List.map (fun p -> p, Variable p) parameters
```

```
5           @ store
6
7       let rbody = reduce body context bodyStore
8
9       let cl =
10          Constant <| Closure(name, parameters, rbody, [])
11
12      ((name, cl) :: store)
13      |> reduce expr2 context
```

Reducing ADT definitions happens in listing 1.27 by adding the constructors to the store and continue reducing the following expression. If the constructor has no parameters, it is an ADT value and otherwise an ADT constructor. A simpler representation of ADT constructors would be desirable, as the differences causes complexity in the patterns matching.

Listing 1.27. Reducing ADT declarations

```
1   | ADT (adtName, (constructors: (string * Type list) list),
        expression) ->
2       let eval constrDecl =
3           match constrDecl with
4           | name, [] ->
5               (name, Constant <| ADTValue(name, adtName, []))
6           | name, types ->
7               (name, Constant <| ADTClosure((name, types), adtName
                    , []))
8
9       let storeWithConstructors =
10          List.fold (fun s c -> eval c :: s) store constructors
11
12      reduce expression context storeWithConstructors
```

Function application in listing 1.28 are reduced by first reducing the expression to apply and the arguments given. The reduced expression may either be a closure, ADT closure or a constant. When reducing a closure, we check if the context is true, meaning that are not in a dynamic conditional, and thus may unfold function calls. That is, we replace the function application with the specialized function body of the closure. If the context is false, we do not unfold the function body, but rather return and apply the expression. This results in not being able to handle partial function application, which is no problem as this suffices for the examples at hand. If the closure is an ADT closure, we check if all required arguments are given. If the arguments are all static, we return a constant ADT value, and otherwise return an `Apply` to the ADT closure.

Listing 1.28. Reducing function application

```
1   | Apply (f, args) ->
```

```
2       let func = reduce f context store
3       let reduceArgs = List.map (fun a -> reduce a context store)
4
5       match func with
6       | Constant (Closure (name, pars, body, _)) ->
7           if (pars.Length <> args.Length) then fail "Partial
                application not implemented"
8           else if (context) then
9               let nameArgPairs = reduceArgs args |> List.zip pars
10              nameArgPairs @ store |> reduce body context
11          else
12              let args = reduceArgs args
13              Apply(func, args)
14
15      | Constant (ADTClosure ((name, argTypes), adtName, [])) ->
16          List.zip argTypes args |> ignore
17          let args = reduceArgs args
18
19          if allStatic args
20          then Constant(ADTValue(name, adtName, getValues args))
21          else Apply(func, args)
22
23      | Constant c -> Constant c
24      | _ ->
25          sprintf "Reduced func not a function: %O" func
26          |> fail
```

The complexity of the reducer lays in the reduction of patterns. The reduction of patterns is given in listing 1.29, but which utilizes the `match1` function given in listing 1.30. The `match1` will try to match a single pattern with the match expression. When a pattern contains a variable name, it is a binding of an expression to a name, which shall be used in the body of the patterns. Thus, this is the main task of the `match1` function: Try to match a value with a given pattern and collect all the bindings in the pattern. This is the complex and main part of this reduction. A matching is represented by the type `ReducedMatch`, which can take three different forms. Either it is a `NoMatch`, if it is certain that the pattern will never match, even when the dynamic variables of the match expression is known. A `DynamicMatch` means that the pattern could possibly match, once the dynamic variables are known. The `StaticMatch` is certain that the pattern will match. Dynamic and static matches carry a list of bindings, which is a list of tuples of name and expression.

The `matchMany` function is a helper function for when many expressions determines whether a match is static or dynamic or a no match - e.g. in lists. The function matches all expressions in the list, and checks if one of them is a `NoMatch`. Otherwise it collects all the bindings that result from the matching. If

any of them is dynamic, then the entire result is a dynamic match, and otherwise it is static.

The last helper function is `collect` which simply traverses the pattern and collects all variables as bindings to itself. This is used when the match expression is itself a dynamic variable. Then we cannot match any further in the pattern, but we still need to collect the rest of the bindings from the pattern in order to reduce the body.

Returning back to the reduce function in listing 1.29. First the match expression is reduced, and the result is either a constant or it is dynamic. In either case we need to match the expression with the given patterns and reduce the body of the pattern. If the match expression is constant, we know that one of the patterns must match statically. Thus, we just need to find the first static match. The bindings are then added to the store, before reducing the body of the pattern and returning the reduced expression.

If the match expression is dynamic we need to do the same thing, we check all patterns and choose the ones that matches. If the pattern is a match, the bindings are added to the store, and the body is reduced. The output is a tuple together with a boolean flag communicating whether the binding was static or dynamic. If the first match in the list is static, we know that this will always match and thus we can simply return the body of that pattern. Otherwise we return a pattern expression.

An important thing to note is that the patterns, like conditionals, control the flow of the program. Therefore, according to our unfolding strategy, the context must be switched to false inside dynamic pattern matching, to avoid infinite unfolding.

Listing 1.29. Reducing pattern match expressions

```
1  | Pattern (matchExpression, patternList) ->
2      let rActual = reduce matchExpression context store
3      match rActual with
4      | Constant v ->
5          patternList
6          |> List.pick (fun (pat, body) ->
7              let m = match1 (rActual, pat)
8              match m with
9              | StaticMatch bindings -> Some(reduce body context
                   <| (bindings @ store))
10             | _ -> None)
11     | _ ->
12         let reducedMatches =
13             patternList
14             |> List.choose (fun (pat, body) ->
```

```
15              let m = match1 (rActual, pat)
16              match m with
17              | StaticMatch bindings -> Some((pat, reduce body
                    context <| (bindings @ store)), true)
18              | DynamicMatch bindings -> Some((pat, reduce
                   body false <| (bindings @ store)), false)
19              | NoMatch -> None)
20
21        match reducedMatches with
22        | ((_, body), true) :: _ -> body
23        | [] ->
24            raise
25            <| ReduceError(e, "All patterns are known statically
                    to not match")
26        | many -> Pattern(rActual, List.map fst many)
```

Listing 1.30. Function for matching an expression with a pattern expression

```
1
2  type Bindings = list<string * Expr>
3
4  type ReducedMatch =
5      | NoMatch
6      | DynamicMatch of Bindings
7      | StaticMatch of Bindings
8
9  (* If actual is dynamic, this traverses the pattern
10    and collects all variables as bindings *)
11 let rec collect =
12     function
13     | Variable x -> [ (x, Variable x) ]
14     | Tuple (e1, e2) -> collect e1 @ collect e2
15     | List exprs -> exprs |> List.collect collect
16     | ConcatC (h, t) -> collect h @ collect t
17     | Apply (name, values) -> values |> List.collect collect
18     | _ -> []
19
20 (* match1 returns
21        None if there can never be a match
22        Some bindings *)
23 let rec match1 (actual: Expr, case: Expr) =
24     match (actual, case) with
25     | Constant v1, Constant v2 when v1 = v2 -> StaticMatch []
26     | _, Constant (CharValue '_') -> StaticMatch []
27     | _, Variable x -> StaticMatch [ (x, actual) ]
28     | Variable a, _ -> DynamicMatch <| collect case
```

```
29      | Tuple (e1, e2), Tuple (p1, p2) ->
30          matchMany [ e1; e2 ] [ p1; p2 ]
31      | List es, List pats when es.Length = pats.Length ->
            matchMany es pats
32      | List (h :: t), ConcatC (h', t') ->
33          matchMany [ h; (List t) ] [ h'; t' ]
34      | Apply (name, exprs), Apply (pname, pats)
35          when List.length exprs = List.length pats ->
36          matchMany (name :: exprs) (pname :: pats)
37      | Apply (name, exprs), Constant (ADTValue (x, _, vals))
            when List.length exprs = List.length vals ->
38          matchMany
39              (name :: exprs)
40              (Variable x :: List.map Constant vals)
41      | Constant (ADTValue (name, _, [])), Constant (ADTValue (x,
            _, [])) when x = name -> StaticMatch []
42      | Constant (ADTValue (name, _, values)), Apply (Variable
            name1, exprs) ->
43          if name = name1
44          then matchMany (List.map Constant values) exprs
45          else NoMatch
46      | _, _ ->
47          NoMatch
48
49  and matchMany exprs pats =
50      let matches = List.map match1 <| List.zip exprs pats
51      if List.exists ((=) NoMatch) matches then
52          NoMatch
53      else
54          let bindings =
55              matches
56              |> List.collect (function
57                  | DynamicMatch bindings -> bindings
58                  | StaticMatch bindings -> bindings
59                  | NoMatch -> failwithf "Impossible: NoMatch")
60
61          if List.exists (function
62              | DynamicMatch _ -> true
63              | _ -> false) matches then
64              DynamicMatch bindings
65          else
66              StaticMatch bindings
```

2.4 Discussion

Complexity As can be seen in listing 1.30, there are many cases which need to be considered. The complexity is mainly caused by the different representations of ADTs in the abstract syntax. These are either represented as *ADTValues*, *ADTClosures* or function applications *Apply*. More complexity is added from the different representation of lists, in the form of Lists and ConcatC. This is unfortunate since the reducer has to be clever and extremely precise, in order to work correctly and optimally. Unfortunately this complexity has prevented us from being able to reduce the complex views containing several nested structures. It has however been possible to reduce on smaller examples, which is enough to showcase the point we want to make.

Program point specialization Another improvement is that of *program point specialization*, which as mentioned is one of the three main techniques in partial evaluation that we do not exploit. In a functional language, program points are the names of the functions. Essentially program point specialization is to define and memorize specialized functions. This was exemplified in 1.20 where we defined a new function `updateIncrement`, which can then replace all expressions in the program where `update Increment model:dyn` is called (here :dyn denotes that the model is unknown, i.e. dynamic). Thus, one function may appear many places in the program in a specialized version with a specialized name, and program point specialization is a combination of defining and folding functions.

The technique would be implemented as follows. Whenever a function call is reduced, we memorize the name and arguments. When the same function call is encountered in a different place in the program (e.g. `update Increment model:dyn` may be called different places in the program) we lookup if the reduction has been computed previously. If so, we can use a copy of that. If not, the function has not been called with that combination of static and dynamic arguments before, and thus we reduce it now and memorize the reduction.

Currently we are not utilizing program point specialization in the reducer, we simply reduce every time we encounter a function call (and we are not within a dynamic if-branch). This means that we may specialize the same function body with the same arguments many times. Clearly memorizing the reduction would be beneficial in terms of efficiency.

3 Model-view-update architecture in JaLi

With the JaLi language in place, this chapter will show how the MVU architecture can be implemented using it. The button example will be continued, which is influenced by the Elm button example.

The program will have a model, that holds the current state of the program. The functions and types are expressed in listing 1.31. An update function defining how to handle each possible action, like *Increment* and *Decrement*. The last function *view* is responsible of taking the model's value and transforming it to HTML for the browser to render. An ADT named `Msg` will hold the different actions that are possible. The HTML structure can be represented by a `Node` type, which was explained in chapter 1. The `view` function takes the model and places into the `Tag` structure that is going to be the HTML in the end but returns here just a structure of type `Node`. This can then be passed to `viewToHtml` which translates a value of type `Node` into a string of valid HTML.

Listing 1.31. Pseudo code to show functions and types

```
type Node
type Msg
update: Msg -> Model -> Model
view: Model -> Node

type Differ
diff: Node -> Node -> Differ
patch: Differ -> Node -> Node

patchToJs: Differ -> String
viewToHtml: Node -> String
```

An example implementation was shown in chapter 1, but is presented again here for the ease of the reader.

Listing 1.32. Button example written in JaLi

```
type Node = Text String | Tag String [Tuple] [Node];
type Msg =
    Increment
  | Decrement;

model = 0;

func update msg val =
  match msg with
    | Increment -> val + 1
    | Decrement -> val - 1
end

```

```
14 func view val =
15   Tag ('div') [('class', 'body')] [
16     Tag ('button') [('onClick', update (Increment) (
           val) )] [Text ('Increment')],
17     Tag ('p') [] [Text (val)],
18     Tag ('button') [('onClick', update (Decrement) (
           val) )] [Text ('Decrement')],
19   ]
20 end
21
22 view model
```

This implementation is enough to prepare one static view that can be rendered in the browser. By applying `viewToHtml` on the node we get the HTML that has been saved to a file and displayed in the browser shown in figure 2.

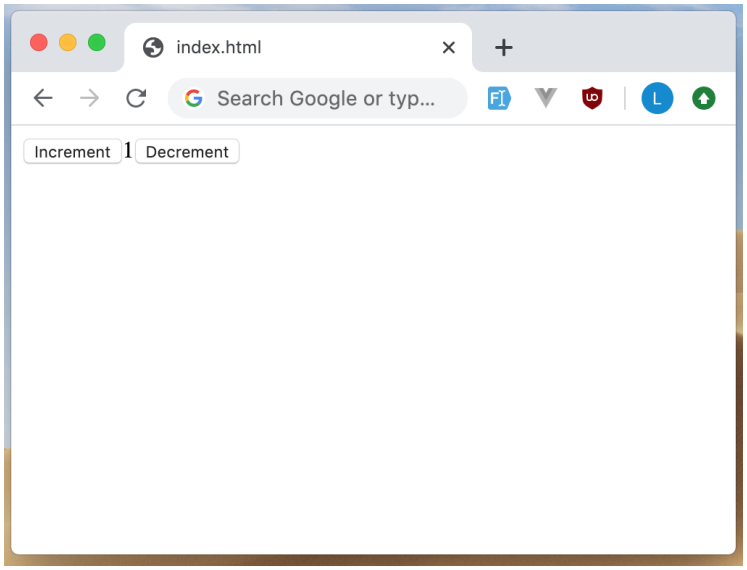

Fig. 2. Initial view based on the JaLi implementation

To make the page dynamic it needs a *handler*, attached to all the places where a user can interact with the website. For the button program this means to attach an on-click handler, that is triggered when the button is being clicked. The JaLi example listing 1.32 shows that `update Increment` and `update Decrement` has been attached to the buttons. However, to make them actually work in the browser, all these actions need to be written in JavaScript, the only dynamic language the browser understands. This can be achieved by implementing a JaLi to JavaScript compiler, that translates the JaLi code into JavaScript. This

has not been implemented, but it is very much possible, and thus we will simply simulate it by manual translation.

For the button example the actions *Increment* and *Decrement* alter the displayed value, by either incrementing its value or decrementing by one. To make this as generic as possible and adapt previous ideas like VDOM comparing, two functions will be implemented: `diff` and `patch`. These are also the functions that need to be translated to JavaScript and executed—with the update function—on every action. The `diff` function detects changes between two views and the `patch` function takes these changes and patches them in the view that is being shown.

Constructing two different views in listing 1.33 by calling the `view` function with 1 and 2 would—when given to the `diff` function—return the `Differ` from listing 1.34.

Listing 1.33. Two views to compare

```
view1 = Tag ('div') [] [
    Tag ('button') [] [Text ('Increment')],
    Tag ('p') [] [Text (1)],
    Tag ('button') [] [Text ('Decrement')],
];
view2 = Tag ('div') [] [
    Tag ('button') [] [Text ('Increment')],
    Tag ('p') [] [Text (2)],
    Tag ('button') [] [Text ('Decrement')],
];
```

Listing 1.35 shows the definition for the `diff` function. It receives two views and traverses through it, comparing always the first element in the list of nodes. If both `Tags` are the same it continues comparing, by using the helper function `fold` to iterate over the children nodes on the compared tags. If it encounters a difference either on the `Tag` or on the `Text` it returns a `Change` constructor with the value from view2, since view2 is the newer view. `fold` just takes the first element out of the list and calls `diff` on it again. If that call returns a `Change` it indicated that there is a change at that child by returning a `Path` with the index of the child. As a result, we get a `Path` to the child that must be changed. This function at this stage only detects the first difference on two views and then terminates. This is for the trivial button example sufficient but should be extended for real use.

Listing 1.34. Differ path detecting change on `Text` node

```
$ Constant
(ADTValue
   ("Path","Differ",
    [IntegerValue 1;
     ADTValue
       ("Path","Differ",
```

```
7          [IntegerValue 0;
8           ADTValue
9             ("Change","Differ",
10              [ADTValue ("Text","Node",[IntegerValue 2])])
11         ])]))
```

Listing 1.35. Diff function detecting changes on two views

```
1  func diff view1 view2 =
2    func fold nodes1 nodes2 index =
3      match (nodes1, nodes2) with
4      | (n1::r1, n2::r2) ->
5        (match diff (n1) (n2) with
6        | Null -> fold (r1) (r2) (index + 1)
7        | change -> Path (index) (change))
8      | _ -> Null
9    end
10
11   match (view1, view2) with
12   | (Tag (t1) (atts1) (children1), Tag (t2) (atts2) (children2)
         ) ->
13       if t1 == t2
14       then fold (children1) (children2) (0)
15       else Change (Tag (t2) (atts2) (children2))
16   | (Text (s1), Text (s2)) ->
17       if (s1 == s2) then Null else Change(Text s2)
18 end
```

In order to apply this detected change, the `patch` function takes the old view, the changes and returns a new view.

Listing 1.36. Patch function applying changes on old view

```
1  func patch view changes =
2    match changes with
3    | Null -> view
4    | Change (n) -> n
5    | Path (index) (d) ->
6      match view with
7      | Tag (name) (atts) (nodes) ->
8        func f i item =
9          if i == index
10         then patch (item) (d)
11         else item
12       end
13       items = mapi (f) (nodes);
14       Tag (name) (atts) (items)
```

```
15      | _ -> 'Patch exception'
16  end
```

It does this by iterating over the `Differ` ADT held in `changes`, which is a recursive path structure that has a `Path` and `Change` constructor. When a `Change` is encountered it immediately returns the value from it, otherwise it will take the index from the `Path` and find the corresponding `Tag` node in the view and follow it recursively by calling patch again. It uses two helper functions: `mapi` and the inner function called `f`. `mapi` iterates through a list a applies the given function on each element it passes while maintaining an index, that is passed to the function. This allows the tracking of elements in the view and the index from the `Path`. Ultimately `patch` will then return an updated `Node` structure that looks like `view2` from listing 1.33.

By compiling the patch function to JavaScript, it can be executed in the browser, thus updating the right child in the actual view. One missing feature is to update the value of the model stored in the JavaScript in the browser.

Generating JavaScript code from JaLi In order to make the whole MVU cycle work, a few additions needs to happen to the existing program. In principal when a button is clicked, the model needs to be either increased or decreased by one, a new view created with the updated model, the differences between the old and new view calculated and then applied on the old view, by only updating the changed part.

Listing 1.37. Update and patch on action

```
1  func updateAndPatch action =
2    oldModel = readGlobal ('model');
3    newModel = update (action) (oldModel);
4    eval (patchToJs (diff (view newModel) (view (oldModel))))
5    setGlobal ('model') (newModel)
6  end
```

This function when compiled to JavaScript and executed would read from global variable called `model` the value, and update it accordingly to the action passed in. This `newModel` would then be used to construct a new view and compared by using `diff` and the old view. The `update` and `diff` functions are written in JaLi and have been introduced previously. The new and crucial function is the `patchToJs` function. It is in essence the same as the `patch` function but generates optimized JavaScript code to handle the change.

The global reading of the variable is not shown here as this would be generated by the JavaScript compiler and so is the setting of the variable from line 5. Even though the `update` and `diff` functions are implemented they would also need to be compiled to JavaScript. The only part that is generating JavaScript that needs no further compiling is the result from the `patchToJs` function. It would be executed by the native implemented JavaScript function *eval*[3].

[3] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval`

By evaluating the `patchToJs` function with the Path that was previously computed, we can compute the JavaScript needed to update the correct child. The `patchToJs` function elevates the complete DOM access, by using accessors like *children* on DOM nodes to follow the `Differ` path down to the change.

Listing 1.38. Patch function generating JavaScript

```
1  func patchToJs changes =
2    func loop seq chs =
3      match chs with
4      | Null -> 'patchToJs -> No changes detected'
5      | Path (idx) (d) ->
6        loop (append (seq) ([idx])) (d)
7      | Change (n) ->
8        match n with
9        | Text (t) ->
10         (changeInnerHTMLToString (seq) (t))
11       | tag -> changeElementToString (tag) (seq)
12       | _ -> 'error'
13   end
14   loop ([0]) (changes)
15 end
```

Listing 1.39. JavaScript that patches the view

```
1  document.body.children[0].children[1].children[0].innerHTML =
       2;
```

JavaScript compiler Now lacking the Jali to JavaScript compiler does not stop us from compiling the code to JavaScript by hand. In theory and only able to do it manually this section shows how the JavaScript compiled program should look like.

To make the ADT functionality in JavaScript work, a class for the super-type of an ADT could be defined and the different constructors be classes that extend this class, with a constructor and fields for the different type arguments on an ADT.

Listing 1.40. JavaScript code for ADTs and other types

```
1  let model = 0;
2  class Message { }
3  class Increment extends Message { }
4  class Decrement extends Message { }
5  const Inc = new Increment();
6  const Dec = new Decrement();
7
8  class Node { }
```

```
class Tag extends Node {
  constructor(elementName, attributes, children) {
    super();
    this.elementName = elementName;
    this.attributes = attributes;
    this.children = children;
  }
}
class Text extends Node {
  constructor(text) {
    super();
    this.text = text;
  }
}

class Differ { }
class Null extends Differ { }
class Change extends Differ {
  constructor(node) {
    super();
    this.node = node;
  }
}
class Path extends Differ {
  constructor(index, differ) {
    super();
    this.index = index;
    this.differ = differ;
  }
}

class Action { }
class Click extends Action { }
```

The code in listing 1.40 also shows the global model variable.

The key part of the MVU cycle in this case is the `updateAndPatch` function that is attached to the buttons and triggered by the *onclick* event.

Listing 1.41. JavaScript code for updateAndPatch

```
function updateAndPatch(message) {
  oldModel = model;
  newModel = update(message, oldModel);
  eval(patchToJs(diff(view(oldModel), view(newModel))));
  model = newModel;
}
```

All the things previously described from the `updateAndPatch` now need to happen in JavaScript.

The `update` function in JavaScript is as trivial as it was before—just incrementing or decrementing the value.

Listing 1.42. Update function in JavaScript

```
function update(message, model_) {
  if (message instanceof Increment) {
    return ++model_;
  } else {
    return --model_;
  }
}
```

The result will then be used to build the two views, which are then compared.

Listing 1.43. View function in JavaScript

```
function view(model) {
  return new Tag('div', [], [
    new Tag('button', [(new Click(), 'updateAndPatch(new
        Increment())')], [
      new Text('Increment')
    ]),
    new Tag('p', [], [new Text(model)]),
    new Tag('button', [(new Click(), 'updateAndPatch(new
        Decrement())')], [
      new Text('Decrement')
    ]),
  ]);
}
```

Listing 1.44. Diff function in JavaScript

```
function diff(view1, view2) {
  if (view1 instanceof Tag && view2 instanceof Tag) {
    if (view1.elementName === view2.elementName) {
      function fold(nodes1, nodes2, index) {
        if (Array.isArray(nodes1)
          && Array.isArray(nodes2)
          && nodes1.length > 0 && nodes2.length > 0) {
          const tail1 = nodes1.slice(1);
          const tail2 = nodes2.slice(1);
          const change = diff(nodes1[0], nodes2[0]);
          if (change instanceof Null) {
            return fold(tail1, tail2, ++index);
          } else {
```

```
            return new Path(index, change)
          }
        } else {
          return new Null();
        }
      }
      return fold(view1.children, view2.children, 0);
    } else {
      return new Change(new Tag(view2.elementName, view2.
          attributes, view2.children));
    }
  } else {
    return view1.text === view2.text ? new Null() : new Change(
        new Text(view2.text));
  }
}
```

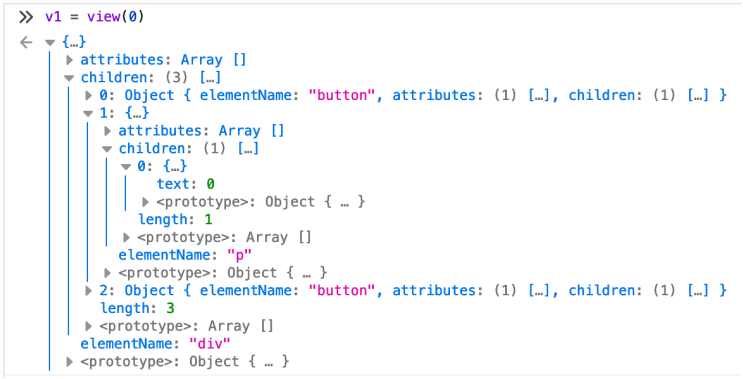

Fig. 3. Constructing a view instance in JavaScript

Finally, the `patchToJs` function applies the changes and forces to rerender the parts in the DOM , because the `patchToJs` is unchanged it returns a string of JavaScript code, that is then executed with the `eval` function.

Listing 1.45. PatchToJs function in JavaScript

```
function changeInnerHTML(seq, value) {
  const element = ((sq) => {
    function loop(str, list) {
      if (list.length === 0) {
        return str;
      } else {
```

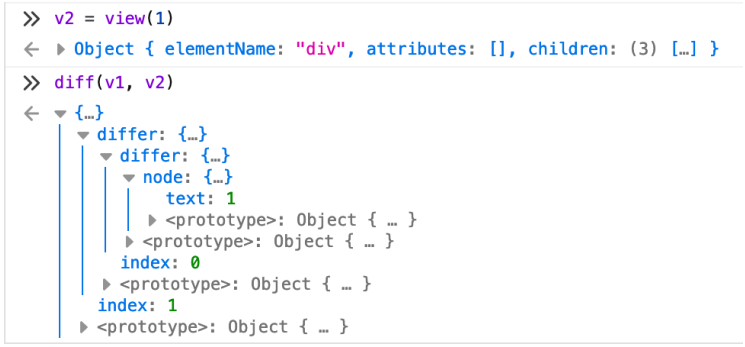

Fig. 4. Differ object after comparing `view(0)` and `view(1)` in JavaScript

```
7         return loop(`${str}.children[${list[0]}]`, list.slice(1)
              );
8       }
9     }
10    return loop('document.body', seq);
11  })(seq);
12  return `${element}.innerHTML = ${value};`;
13 }
14
15 function patchToJs(changes) {
16   function loop(seq, chs) {
17     if (chs instanceof Null) {
18       return 'patchToJs: No changes detected';
19     } else if (chs instanceof Path) {
20       return loop(seq.concat(chs.index), chs.differ);
21     } else {
22       const node = chs.node;
23       if (node instanceof Text) {
24         return changeInnerHTML(seq, node.text);
25       } else {
26         return 'patchToJs: Not implemented: changeElement(node,
              seq)';
27       }
28     }
29   }
30   return loop([0], changes);
31 }
```

Listing 1.46. Eval the JavaScript string that patches the view

```
1 eval("document.body.children[0].children[1].children[0].
      innerHTML = 1;");
```

```
» patchToJs(diff(v1, v2));
← "document.body.children[0].children[1].children[0].innerHTML = 1;"
```

Fig. 5. Result JavaScript string from calling `patchToJs`

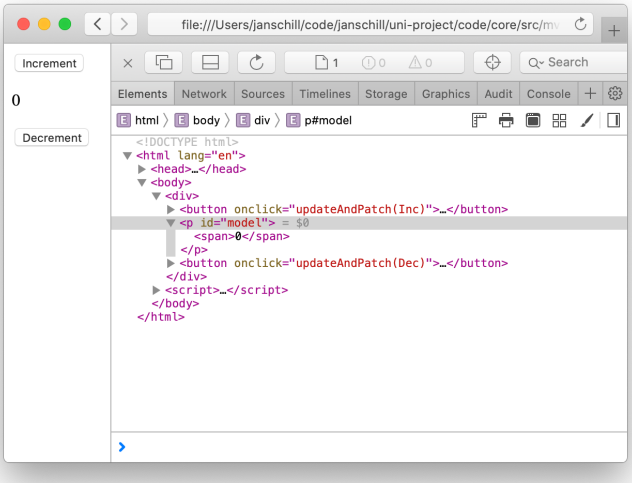

Fig. 6. Initial view

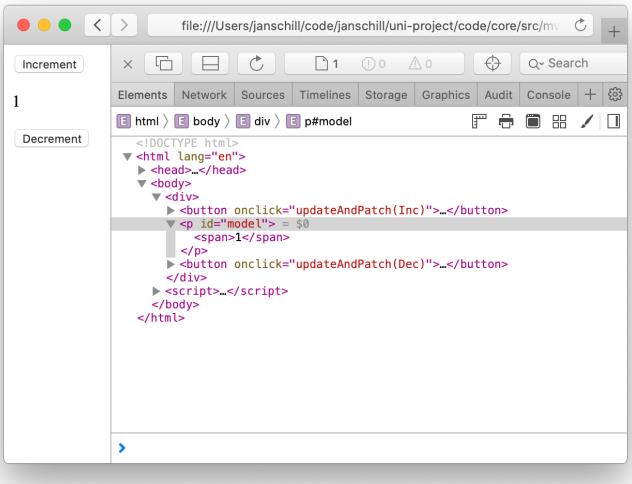

Fig. 7. View after clicking increment button

4 Reducing the model-view-update architecture

The introduction of the MVU pattern in JaLi and implementation of the JavaScript to make it run in the browser, shows that the interaction works. This chapters will now take the reduction strategies from chapter 2 and apply them to the example at hand.

The desired outcome is to have a view with two small JavaScript snippets that sit on the buttons that apply the change directly onto the DOM node, without the need of having to go through the whole cycle of *diffing* again.

When diffing two views, both of those DOM trees need to be traversed completely and compared node by node until it has calculated the full *Differ* path. This should only happen when absolutely necessary or at a time where a user does not notice it, for example during compile time.

The problem is during compile time the value of the model in the button example is unknown. This makes pre-computations difficult. This is however where partial evaluation comes in handy.

It is intuitive to see that the two buttons: increment and decrement, will always only change the model by either increasing or decreasing its value by one. It is also known where this model is being rendered in the DOM tree. This means that the program should not have to through the whole cycle for every action. The path for the children to update will remain the same.

What currently happens in the MVU cycle is that the value to be inserted in a given child is computed. E.g. evaluating the diff function on the two previously generated views computed a Differ that described exactly what value to insert.

This means that the calculation and the cycle needs to take place every time an action occurs.

However, given the previous button example, the path and the change that needs to happen is already known at compile time. The only unknown variable is the model. Thus is seems that partial evaluation can be exploited to compute two specialized functions for each button: one for increment and one for decrement, each of which needs only the model as the last argument. Translating these two functions to JavasScript and injecting them into the browser together with the view, would effectively result in two JavaScript functions that know exactly where to change the DOM, without any diffing. The example of generating this function based on known input was showcased by the result from `pathToJs` function in listing 1.46. Imagine this same function with a dynamic model as below. Achieving such a function and injecting it into the browser would efficiently circumvent the entire MVU cycle that was needed to get the result of `patchToJs`.

Listing 1.47. Eval the JavaScript string that patches the view

```
1 function updateIncrement {
2     document.body.children[0].children[1].children[0].innerHTML
          = m+1;
3 }
```

The two parts that get reduced—and are crucial for performance gain—are the *update* and the *diff* functions. These two function calls would be reduced as there are two different actions:

Listing 1.48. Increment and Decrement diff calls

```
1 diff (view dyn:model) (view (update Increment dyn:model))
2 diff (view dyn:model) (view (update Decrement dyn:model))
```

The reduction of these two calls will yield a partially evaluated function, that will always return the same *Differ* path, and only the actual *Change* constructor from the *Differ* data type, will be different. This means that for all actions that update the model and thus change the view, will be calculated and reduced during compiling to the point that only the value of the to be inserted node is missing, which means no DOM diffing needs to be done during run-time. The result should be a path containing the changes but based on a dynamic value rather than a specific value. The action should be to increment the dynamic value by one, as specified by specializing update with the static value Increment. We now demonstrate the result of reducing the implementation of the MVU framework in listen 1.49.

Listing 1.49. Reducing pattern match expressions

```
1 type Node = Text String | Tag String [(String, String)] [Node];
2 type Message = Increment | Decrement;
3 type Differ =
```

```
4     Null
5   | Change Node
6   | Path Integer Differ;
7
8  (* UPDATE *)
9  func update msg m = (* return a view *)
10   match msg with
11   | Increment -> m+1
12   | Decrement -> m-1
13 end
14
15 (* VIEW *)
16 func view model =
17   Tag ('div') ([]) ([
18       Text (model)
19   ])
20 end
21
22 (* DIFF *)
23 func diff view1 view2 =
24   func fold nodes1 nodes2 index =
25     match (nodes1, nodes2) with
26     | (n1::r1, n2::r2) ->
27       x = diff (n1) (n2);
28       (match x with
29       | Null -> (fold (r1) (r2) (index + 1))
30       | change -> Path (index) (change))
31     | _ -> Null
32   end
33
34   match (view1, view2) with
35   | (Tag (t1) (atts1) (children1), Tag (t2) (atts2) (children2)
         ) ->
36       if t1 == t2
37       then fold (children1) (children2) (0)
38       else Change (Tag (t2) (atts2) (children2))
39   | (Text (s1), Text (s2)) ->
40       if (s1 == s2) then Null else Change(Text s2)
41 end
42
43 func dynamicDiff model =
44   diff (view model) (view (update Increment model))
45 end
46
47 dynamicDiff
```

We demonstrate this on the small example in listing 1.50.

Listing 1.50. A minimal example of a DOM node

```
func view model =
  Tag ('div') ([]) ([
    Text model
  ])
end
```

Listing 1.51. Reduced result from calling diff with a dynamic model

```
Constant
(Closure
 ("dynamicDiff",["model"],
  Apply
  (Constant
   (ADTClosure (("Path",[Int; Typevar "Differ"]),"Differ",[])),
   [Constant (IntegerValue 0);
    Apply
    (Constant
     (ADTClosure (("Change", [Typevar "Node"]),"Differ",[])),
     [Apply
      (Constant (ADTClosure (("Text", [String]),"Node",[])),
      [Prim ("+",Variable "model",Constant (IntegerValue 1))])
     ])]),
[]))
```

Listing 1.51 displays the reduced program, when *diffing* on two views with a dynamic model, like down in listing 1.52. This means that the reducer is able to reduce away all computations that are needed during the compilation of the program, even though it is missing the crucial information about the value of the model. It indicates exactly where the changes need to happen—as it returns the calculated path—and what operation needs to be executed once the model is known. That is, based on the implementation of the JaLi program, we have been able to reduce the diff function with a dynamic model as specified in **??**. Doing this for decrement would achieve the same result but decrementing the model by 1.

Listing 1.52. Dynamisc diffing

```
diff (view model) (view (update Increment model)
```

Knowing where and what needs to happen on the comparison of two different views, *patching* the old view with the changes from *diffing* is the next step. For this the `patchToJs` function is used. The `patchToJs` function returns the value from the *Change* constructor and sets it to the *innerHTML* of the element established by the path. Because the reduced program, does not know the current

value of the model, it is a primitive operation of adding 1 to the model. Therefore, this is exactly what is being then returned from the `patchToJs` function when reduced in listing 1.53.

Listing 1.53. Reduced result from calling patchToJs with the result from dynamic diff

```
1  Constant
2   (Closure
3    ("dynamicDiff",["model"],
4    Prim
5     ("+",
6      Prim
7       ("+",
8        Constant
9         (StringValue
10         "document.body.children[0].children[0].innerHTML = "),
11          Prim ("+",Variable "model",Constant (IntegerValue 1))),
12         Constant (StringValue ";")),[]))
```

This program returns a function called *dynamicDiff* which takes an argument of model dynamically and has the same operation as before on the model, which is then just concatenated with different JavaScript symbols and the path to the DOM node element.

This is the JavaScript snippet that would be attached to the button handler of increment, handling the the calculation of the new model value, when given by the run-time that would be generated by the JavaScript compiler. This is all computable at compile time since most of the data is already known, and with the help of a partial evaluator. With a compiler from JaLi to JavaScript we can then achieve the desired function, that efficiently updates the view, without needing any diffing.

4.1 Further work

The only problem that is left when reducing is now to read the model from the global state in the JavaScript, and updating the model after updating the view as described previously. This is also the reason the model needs to be implemented in the run-time, which can then be passed to the partially evaluated `updateIncrement` and `updateDecrement` functions. The state would be generated by a JavaScript compiler, that would compile the whole JaLi program into JavaScript, this JavaScript compiler is not implemented and was simulated by the example code snippets from chapter 3. Only updating the model is not enough though, since the DOM needs to be patched as well. The *onClick* handler attached to the button essentially calls the function: *updateAndPatch*.

However the idea here has been demonstrated that the diffing can be completely circumventing by partial evaluation of the initial program. The rest left for the JaLi to JavaScript compiler which can efficiently generate the initial model and read and write to the model from global state.

Conclusion

Partial evaluation is a valuable method of gaining major performance gains in a straightforward way, that starts at the compiler level, whereas most of the time optimizations are searched in the coding style of a program or in using clever architectural patterns. With the complete move of comparing tree-structures in the run-time to the compile time and there doing it only once for each possible action, programs can be made significantly more efficient. This was shown by developing a new functional language called JaLi, which implements a compiler that has to limited degree partial evaluation, symbolic execution and constant folding as optimizations. The JaLi language was used to introduce the MVU pattern and show its drawbacks when performing diffing and patching. A reduced example, with ADTs representing a simple HTML structure was then taken and reduced by the compiler. Doing the diffing and patching during compilation for a given action, generating a JavaScript snippet that could be used to change the DOM dynamically.

Reducing the MVU implementation of a JaLi written web program in chapter 4 efficiently showed how partial evaluation may achieve the complete circumvention of the MVU cycle. This happens by precomputing all the diffing at compile time. This would enable considerable performance gains for web frameworks.

References

1. Sestoft, Peter: Programming language concepts. 2nd edn. Springer London Ltd, (2012)