Partial Evaluator for Source §2 - Documentation

Rachit Bansal - A0214350W Niklas Forsstroem: A0208754Y

1 Introduction

The overall objective of a partial evaluator is to make simplifications of the code at compile time in order to speed up execution at runtime. One example of this could be the following scenario:

```
function a(x,y){
tmp = y+3;
return x + tmp;
}
function b(x){
return a(x,0);
}
b(4);
```

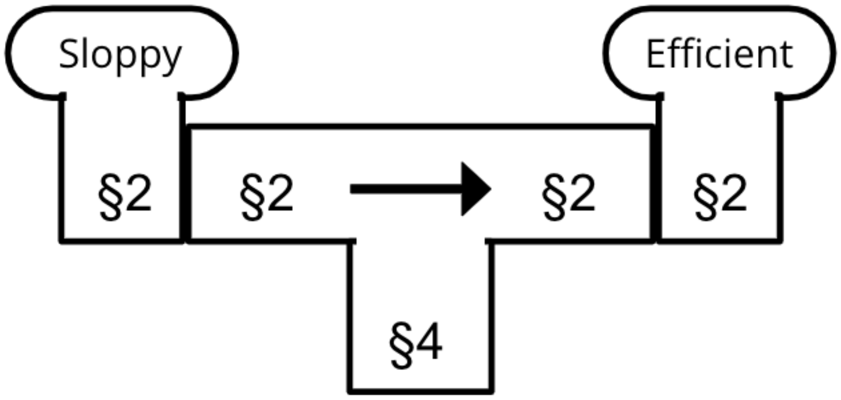
The above code would normally compile the functions separately. The consequence would be that the function a would be invoked every time b is called. This would ultimately result in the variable tmp being re-initialized at every call. One can note that the above program could be equivalently expressed as:

```
function a(x,y){
return x + y+3;
}
function b(x){
return x+3;
}
b(4);
```

The above code however would not cause the variable tmp to be initialized at every call to b(). It would therefore be more efficient at runtime. Even though a() is not explicitly called in our program it would be good practice for the partial evaluator to simplify the expression for a(). We might still invoke a() from the console.

2 Design Choices

The partial evaluator is implemented as a program in Source §4. It accepts the input program and returns an enhanced version. The utilisation of Source §4 allowed the utilisation of the existing infrastructure. Any code which is given as input to the partial evaluator function is treated sequentially (line by line) which



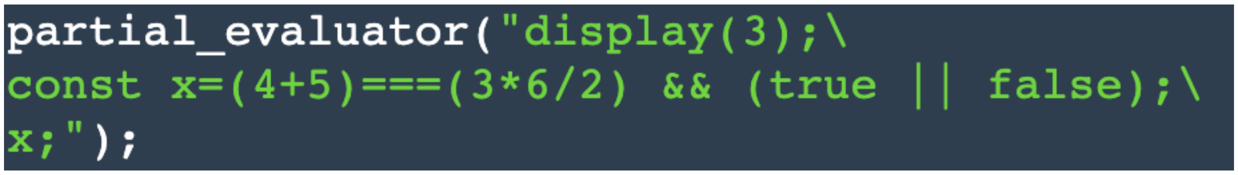
is similar to working of stepper tool. The string final_pro which represents the final program returned at the end gets appended sequentially in eval_seq function with the returned statement/block we get after partially evaluating the corresponding statement/block. Inductively, partial_evaluator(Program + Statement) is equal to partial_evaluator(Program) + partial_evaluator(Statement) carrying the environment of the present block.

2.1 Predefined Constants

Source §2 includes a set of predefined constants. We simply replace them by their value. e.g. math_PI is equal to 3.1415926535897932.

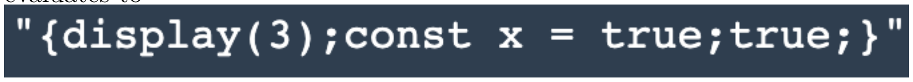
2.2 Primitive Functions

The output of primitive functions are modified to handle the function variables but otherwise it works almost normally. A point to note is that primitive function like display will have to be called in the output program so that it displays when the output program is run. For example,



```
partial_evaluator("display(3);\
const x=(4+5)===(3*6/2) && (true || false);\
x;");
```

evaluates to



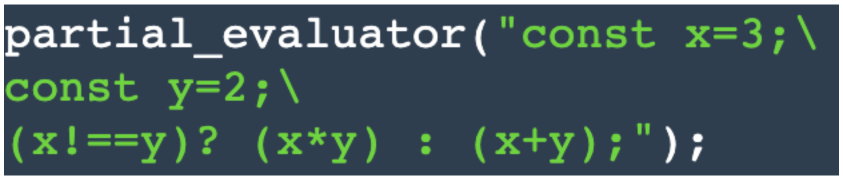
```
"{display(3);const x = true;true;}"
```

2.3 Constant Declarations

On evaluating a statement with constant declaration we evaluate the constant declaration value first. An important point is that the new program must yield an equivalent state. Therefore, free constant/function declarations must be preserved so that when we run the output program we can still access the constant/function from the console. Further, instances of these constants can be replaced throughout the correct scope.

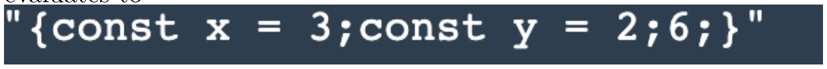
2.4 Conditional Expressions

Conditional expressions which are defined outside of a function body can be simplified in the same way as normal evaluation. For example,



```
partial_evaluator("const x=3;\
const y=2;\
(x!==y)? (x*y) : (x+y);");
```

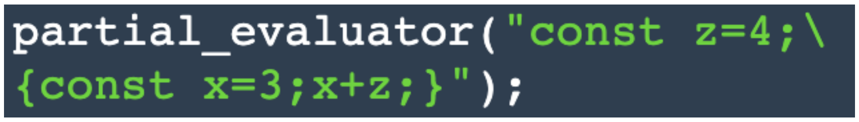
evaluates to



```
"{const x = 3;const y = 2;6;}"
```

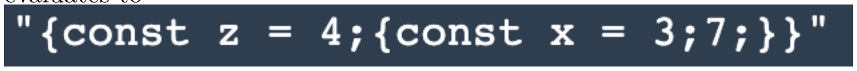
2.5 Blocks

Partial evaluation of a block will sequentially partially evaluate the block and then return the partially evaluated block. An important point is that variables inside a block cannot be accessed outside the block. For example,

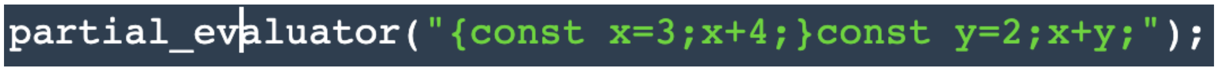


```
partial_evaluator("const z=4;\
{const x=3;x+z;}");
```

evaluates to

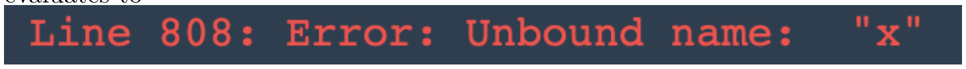


```
"{const z = 4;{const x = 3;7;}}"
```



```
partial_evaluator("{const x=3;x+4;}const y=2;x+y;");
```

evaluates to



```
Line 808: Error: Unbound name:  "x"
```

2.6 Conditional Statements

Conditional statements which are defined outside of a function body return the corresponding partially evaluated block after evaluating the condition. For example,

```
partial_evaluator("const z=3;if(z===2){\
    const w=1;\
    w;\
}\
else{const t=2;\
t+z;}");
```

evaluates to

```
"{const z = 3;{const t = 2;5;}}"
```

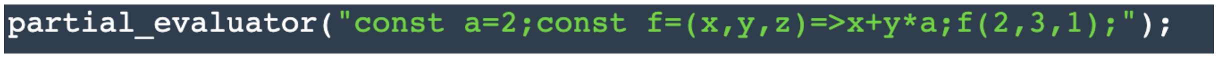
2.7 Function Definitions

A function in Source §2 can only effect the code via returning a value as assignments are not allowed in Source §2. So the best way to simplify a function definition is to represent the function of form "f(x,y,..){...}" in the form of "const f= (x,y,..) => (...);". For example,

```
partial_evaluator("function f(x,y,z){\
const a=2;\
return a*x+y*z;}\
f(1,2,3);");
```

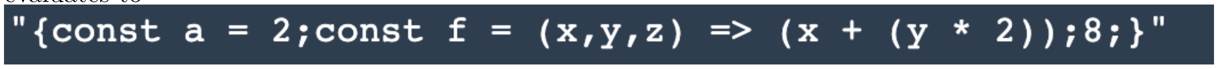
evaluates to

```
"{const f = (x,y,z) => ((2 * x) + (y * z));8;}"
```



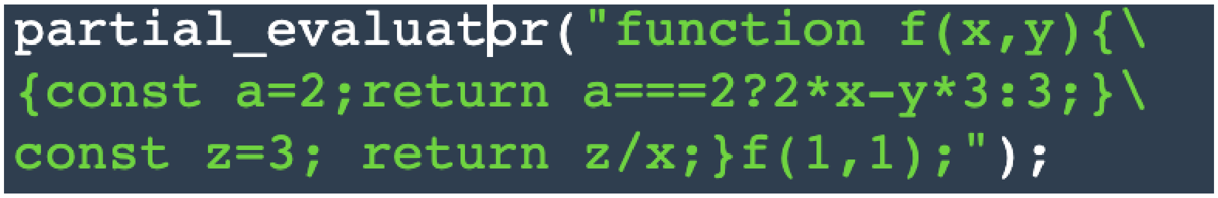
```
partial_evaluator("const a=2;const f=(x,y,z)=>x+y*a;f(2,3,1);");
```

evaluates to



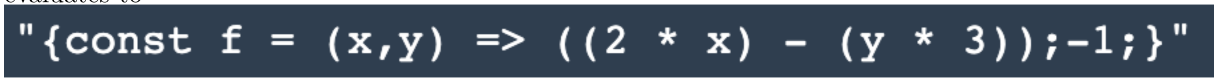
```
"{const a = 2;const f = (x,y,z) => (x + (y * 2));8;}"
```

To simplify the function definition without knowing the arguments of the function definition, we treated the variables as strings and created a new environment with those variables. Defined the primitive functions in such a way such that it can evaluate the mathematical expressions containing the variables. While partially evaluating the function body we return from the evaluation when we hit a return statement. For example,



```
partial_evaluator("function f(x,y){\
{const a=2;return a===2?2*x-y*3:3;}\
const z=3; return z/x;}f(1,1);");
```

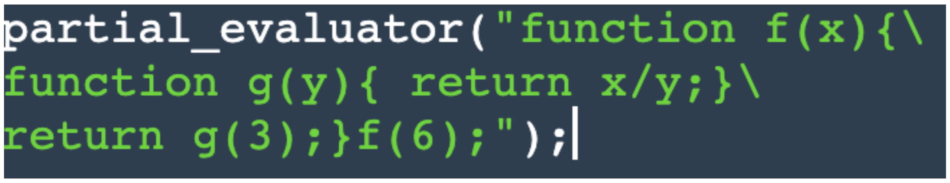
evaluates to



```
"{const f = (x,y) => ((2 * x) - (y * 3));-1;}"
```

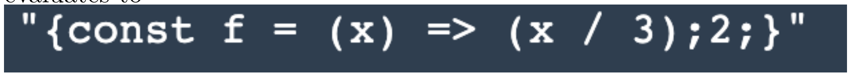
2.8 Function Application

Replace function application by their output by looking up the value we stored for that function and applying it to the given arguments. For example,



```
partial_evaluator("function f(x){\
function g(y){ return x/y;}\
return g(3);}f(6);");
```

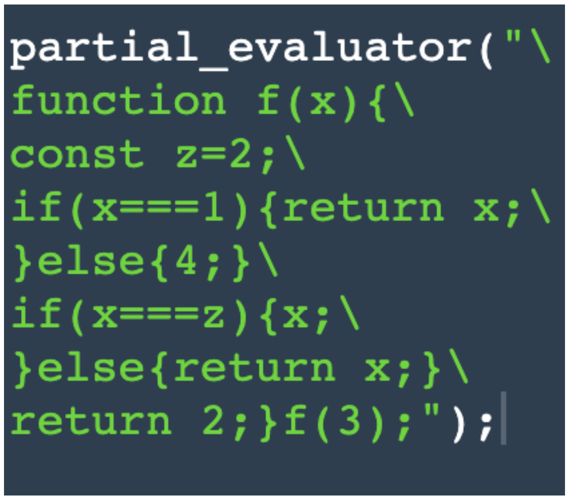
evaluates to



```
"{const f = (x) => (x / 3);2;}"
```

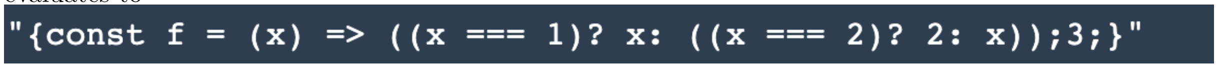
2.9 Function definition containing Conditional Statements

When there are conditional statements we convert it into the form of pred?cons:alt so that we can still write in the form of "const f= (x,y,..) => (...);". We handle the cases in which there is a return statement in if block and there is not a return statement in the else block or vice-versa by using the result of subsequent statements in the block which does not have the return statement. For example,



```
partial_evaluator("\
function f(x){\
const z=2;\
if(x===1){return x;\
}else{4;}\
if(x===z){x;\
}else{return x;}\
return 2;}f(3);");
```

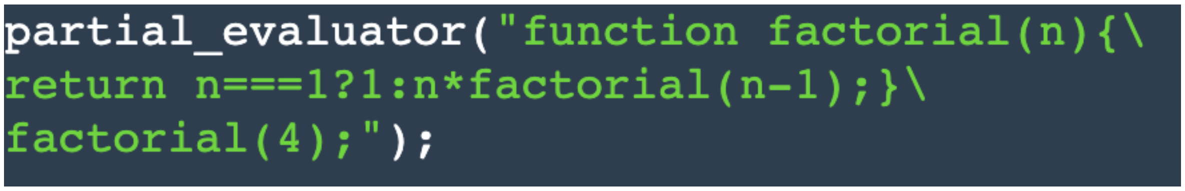
evaluates to



```
"{const f = (x) => ((x === 1)? x: ((x === 2)? 2: x));3;}"
```

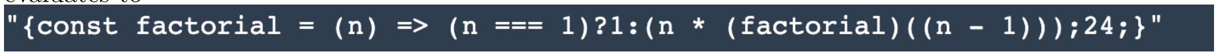
2.10 Recursive function calls

A recursive function call has not been evaluated to find a definite mathematical expression as not all types of recursive functions can be expressed in the form of a specific mathematical expression.



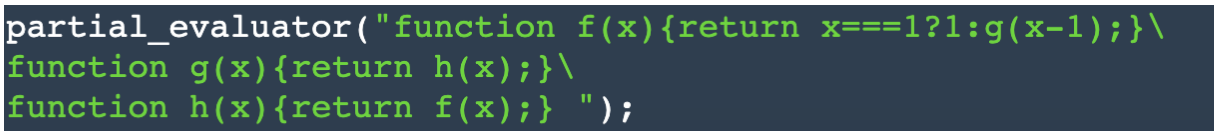
```
partial_evaluator("function factorial(n){\
return n===1?1:n*factorial(n-1);}\
factorial(4);");
```

evaluates to



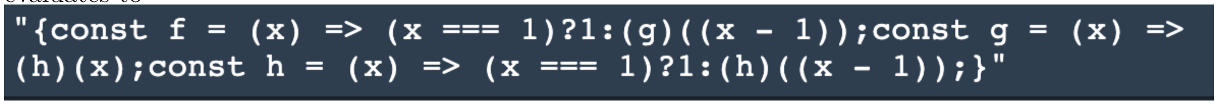
```
"{const factorial = (n) => (n === 1)?1:(n * (factorial)((n - 1)));24;}"
```

As we can see in this example in the code we return we do not replace factorial(n-1) further. As we do not know 'n' we do not know until when we will have to keep on replacing. factorial(4) is calculated properly.



```
partial_evaluator("function f(x){return x===1?1:g(x-1);}\
function g(x){return h(x);}\
function h(x){return f(x);} ");
```

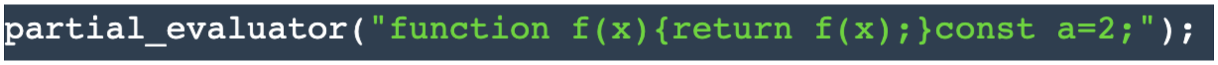
evaluates to



```
"{const f = (x) => (x === 1)?1:(g)((x - 1));const g = (x) =>
(h)(x);const h = (x) => (x === 1)?1:(h)((x - 1));}"
```

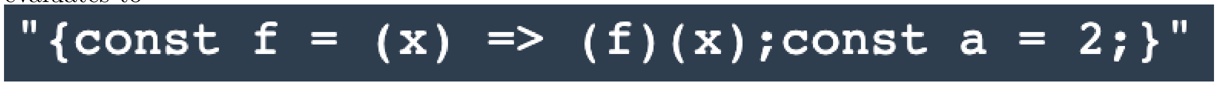
2.11 Time complexity and Termination

As partial evaluation takes place during compile time, the compile time will increase but it will still be less than the execution time without partial evaluation. After compilation, we can call the functions/constants in the program as many times we want from the console and if we use partial evaluator the execution time will be less than usual. To take care about the termination of the partial evaluator, we should initially pass the program through the infinite loop detector so that it can detect any infinite loop. After checking that it doesn't have any infinite loop we can pass the program through the partial evaluator which will surely terminate as the input program terminates. While partially evaluating the input program the partial evaluator will never get stuck in an infinite loop as we have already checked for it using the infinite loop detector. For example,



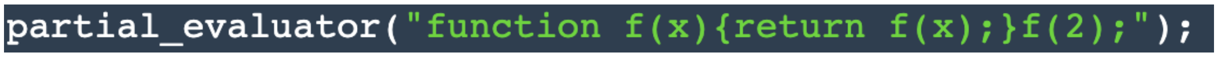
```
partial_evaluator("function f(x){return f(x);}const a=2;");
```

evaluates to



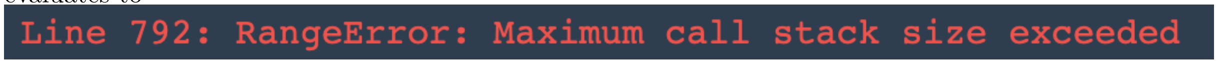
```
"{const f = (x) => (f)(x);const a = 2;}"
```

but



```
partial_evaluator("function f(x){return f(x);}f(2);");
```

evaluates to



```
Line 792: RangeError: Maximum call stack size exceeded
```

because call to the function leads to infinite loop and we should detect such loops before passing such program to the partial evaluator

3 Final Outcome

The overall objective of this project was to implement a partial evaluator for Source §2 that make simplifications of the input code at compile time in order to speed up execution at runtime. The implementation meets the specification as the implemented partial evaluator takes care of the different cases of Source §2 as discussed and returns the required partially evaluated output in a reasonable time.