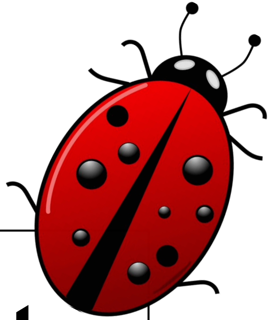
Coccinelle For Rust



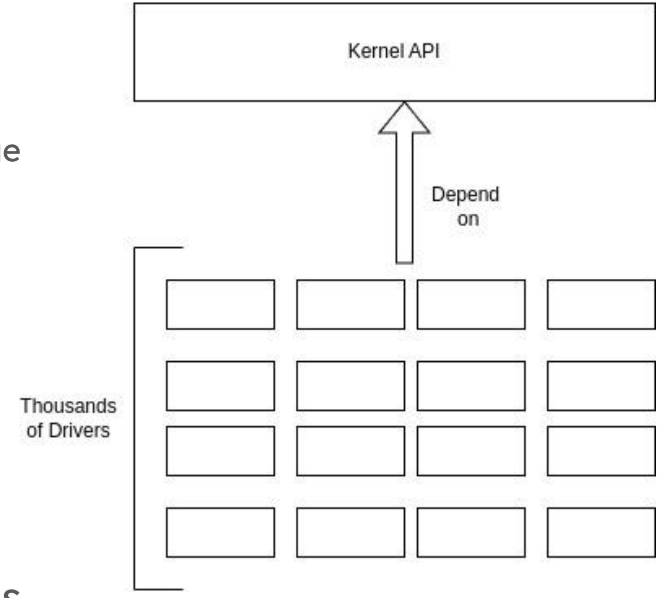
Tathagata Roy, Julia Lawall

What is Coccinelle?

1. Performs repetitive transformations at a large scale
   a. Rust is 1.6 MLOC
   b. Linux Kernel is 23 MLOC
   c. Collateral evolution - a change in the main API leads to change in all clients

2. Provide a transformation language for expressing these changes

3. Changes + Developer Familiarity = (semantic) patches



Latest Developments

1. Addition of the **CTL-VW** engine. Which is the same engine as Coccinelle For C.

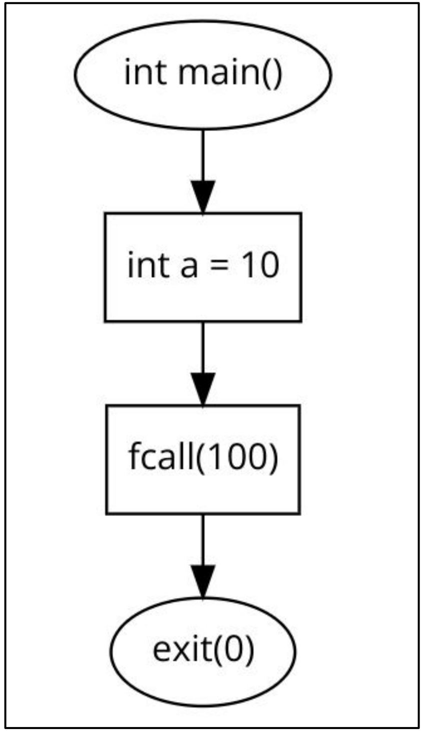
    a. Gives us a standard way (Computation Tree logic formulas) to represent complex control flow paths

    b. C control flow is simple. Conditional nodes only in the function level or inside other conditionals (for the most part)

    c. Rust, not so much

C CFGs

```
int main() {
    int a = 10;
    fcall(100);

    return 0;
}
```



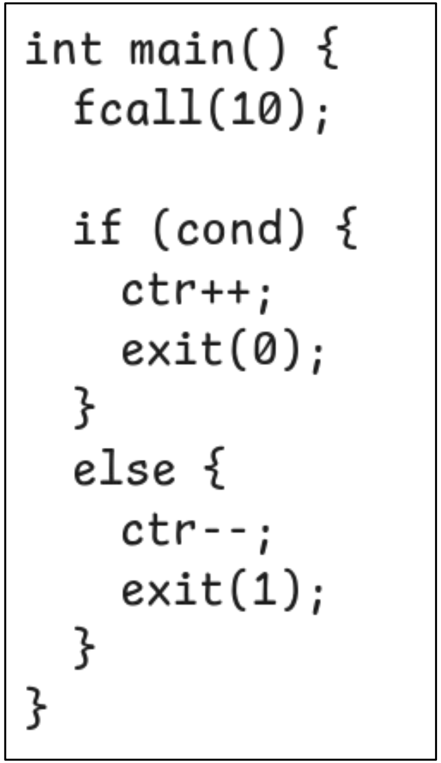
int main()

int a = 10

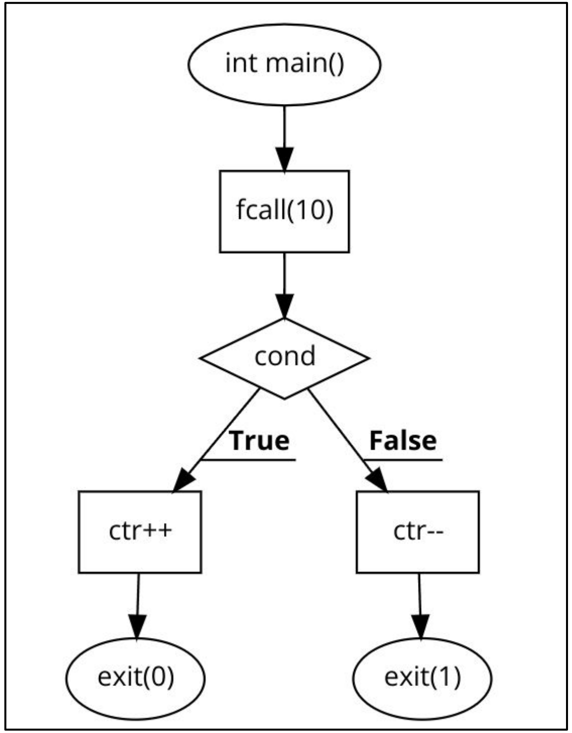
fcall(100)

exit(0)

C CFGs

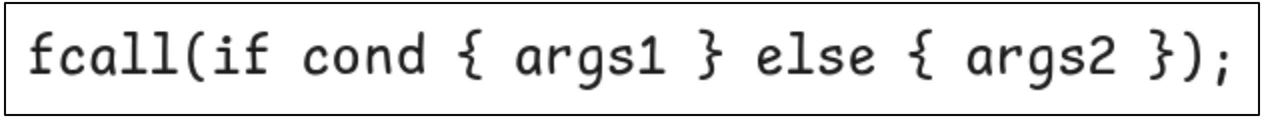


```
int main() {
  fcall(10);

  if (cond) {
    ctr++;
    exit(0);
  }
  else {
    ctr--;
    exit(1);
  }
}
```

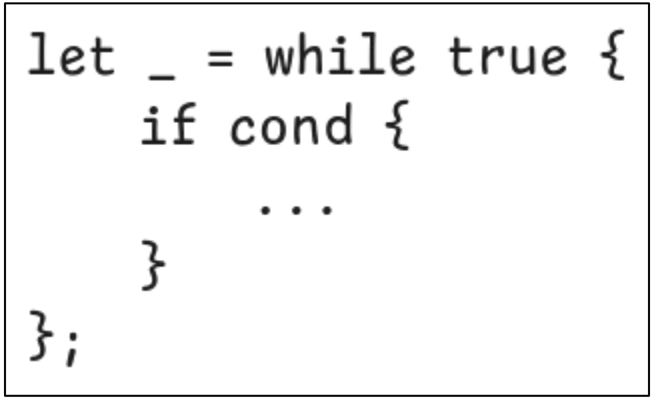


Rust CFG

In rust, if and while/loop statements are expressions. Therefore a control flow branch/loop can occur anywhere.

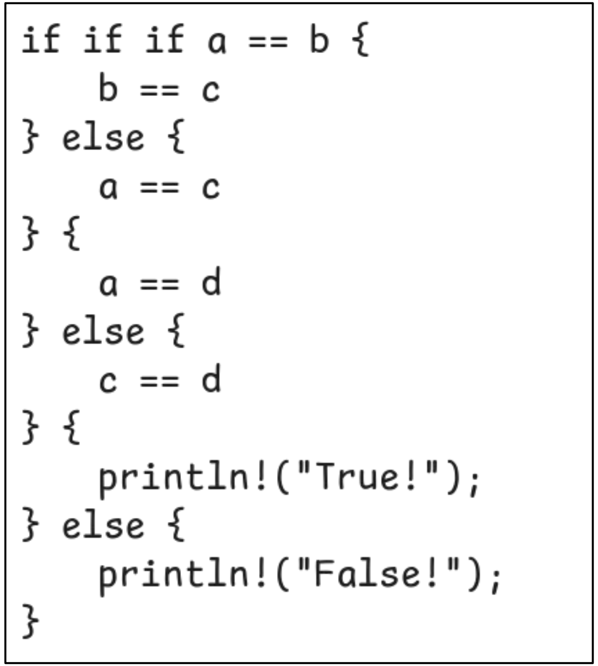


```
fcall(if cond { args1 } else { args2 });
```



```
let _ = while true {
    if cond {
        ...
    }
};
```

Rust Madness



```
if if if a == b {
    b == c
} else {
    a == c
} {
    a == d
} else {
    c == d
} {
    println!("True!");
} else {
    println!("False!");
}
```

unearthly
control
flow

Snippet source: u/ZZaaaccc

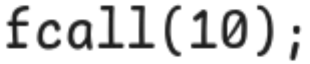
Rust CFG

How to represent rust CFG from the Rust AST?

Rust CFG

How to represent rust CFG from the Rust AST without remaking the compiler?

Node-by-node approach



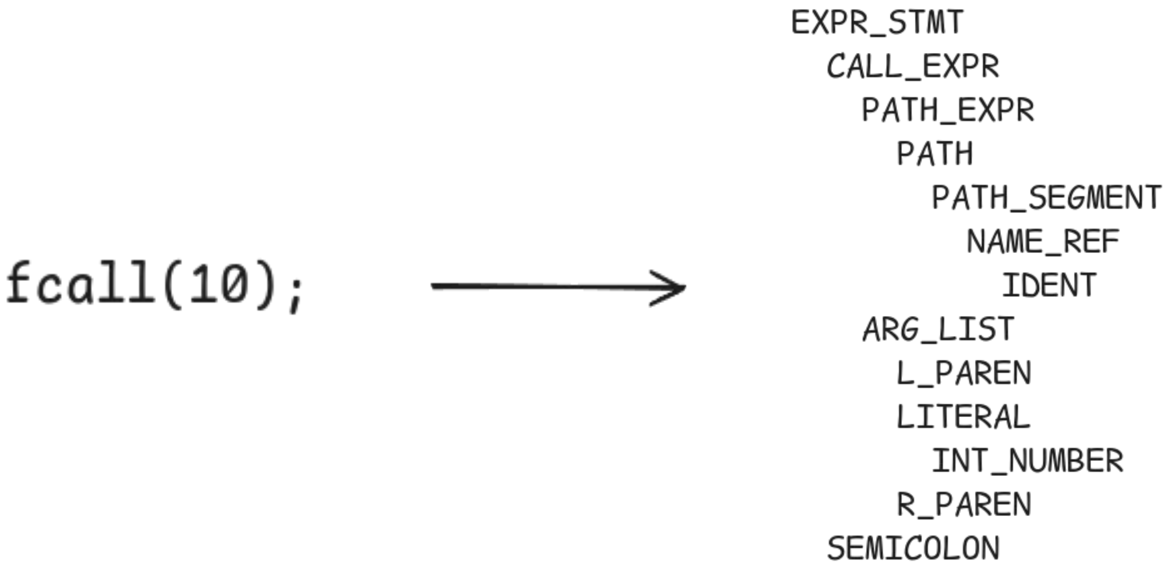
```
fcall(10);
```

Rust CFG

How to represent rust CFG from the Rust AST without remaking the compiler?

Rust CFG

How to represent rust CFG from the Rust AST without remaking the compiler?



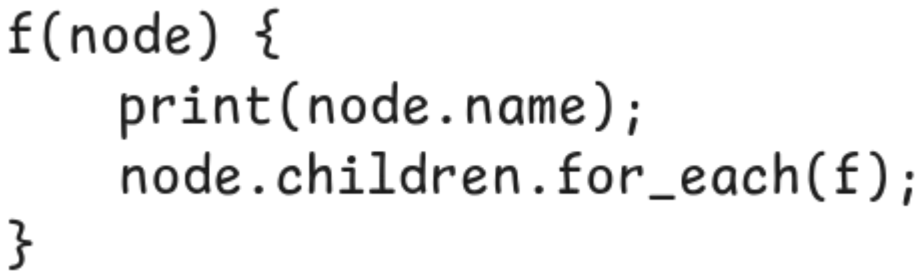
```
fcall(10);
```

```
EXPR_STMT
  CALL_EXPR
    PATH_EXPR
      PATH
        PATH_SEGMENT
          NAME_REF
            IDENT
    ARG_LIST
      L_PAREN
      LITERAL
        INT_NUMBER
      R_PAREN
  SEMICOLON
```

Rust CFG

How to represent rust CFG from the Rust AST without remaking the compiler?

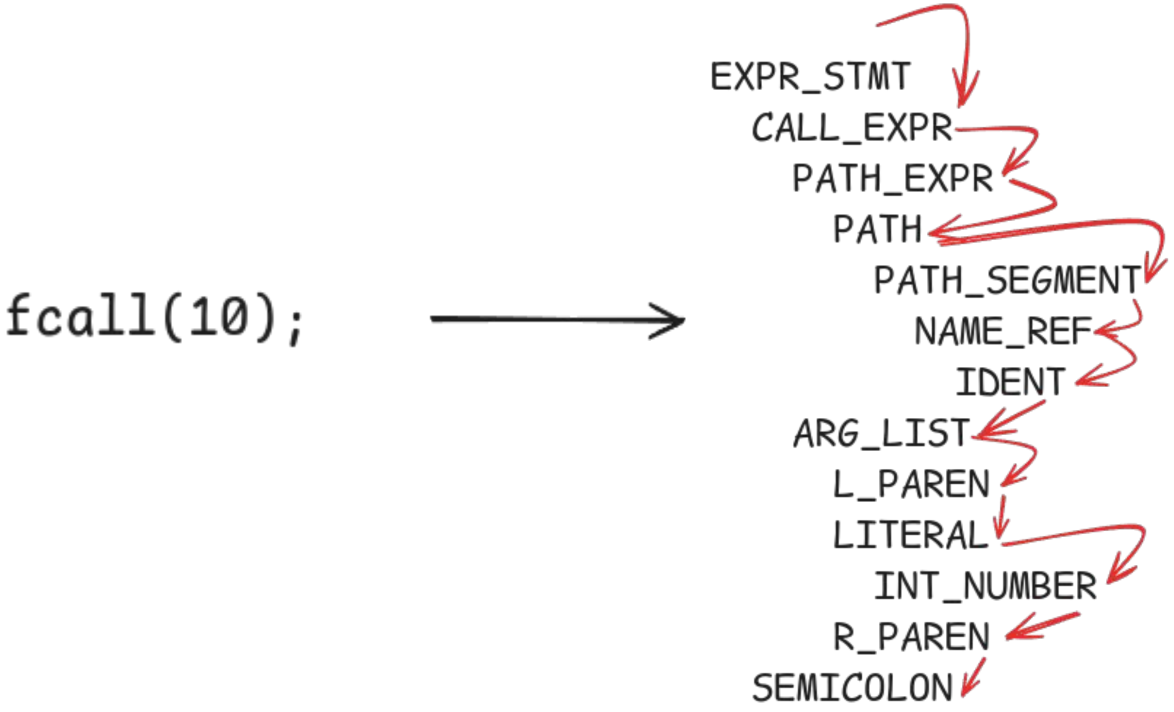
For simple non-branching nodes:-



```
f(node) {
    print(node.name);
    node.children.for_each(f);
}
```

Rust CFG

How to design a Rust CFG from the Rust AST without making remaking the compiler?

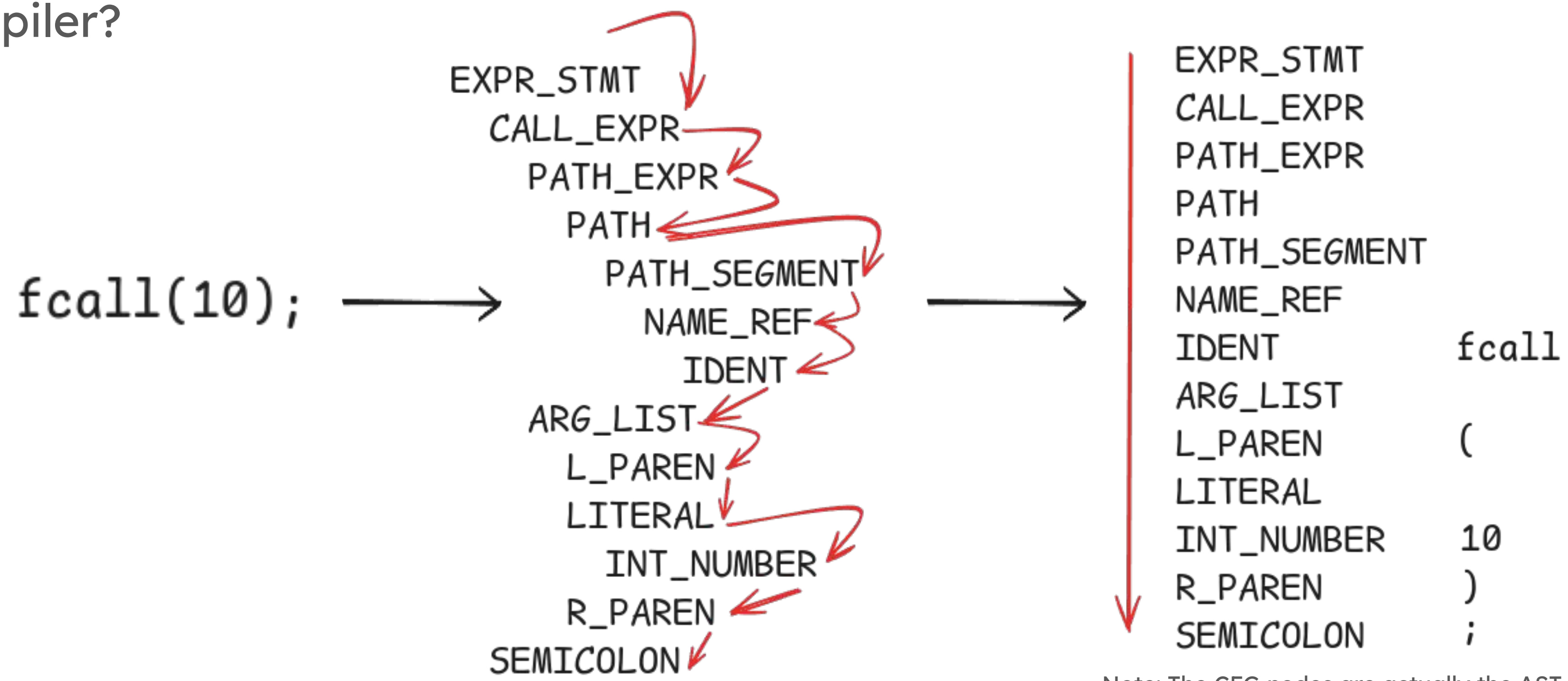


```
fcall(10);
```

EXPR_STMT
CALL_EXPR
PATH_EXPR
PATH
PATH_SEGMENT
NAME_REF
IDENT
ARG_LIST
L_PAREN
LITERAL
INT_NUMBER
R_PAREN
SEMICOLON

Rust CFG

How to design a Rust CFG from the Rust AST without making remaking the compiler?



```
fcall(10);
```

EXPR_STMT
CALL_EXPR
PATH_EXPR
PATH
PATH_SEGMENT
NAME_REF
IDENT
ARG_LIST
L_PAREN
LITERAL
INT_NUMBER
R_PAREN
SEMICOLON

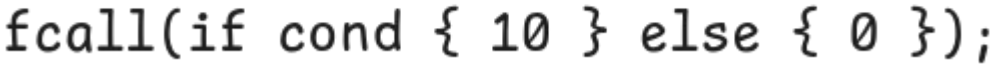
EXPR_STMT	
CALL_EXPR	
PATH_EXPR	
PATH	
PATH_SEGMENT	
NAME_REF	
IDENT	fcall
ARG_LIST	
L_PAREN	(
LITERAL	
INT_NUMBER	10
R_PAREN	)
SEMICOLON	;

Note: The CFG nodes are actually the AST types of the tree

Rust CFG

How to design a Rust CFG from the Rust AST without making remaking the compiler?

But what about branching instructions?

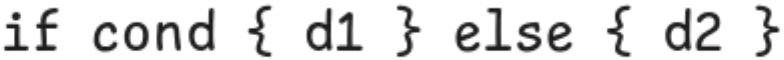


```
fcall(if cond { 10 } else { 0 });
```

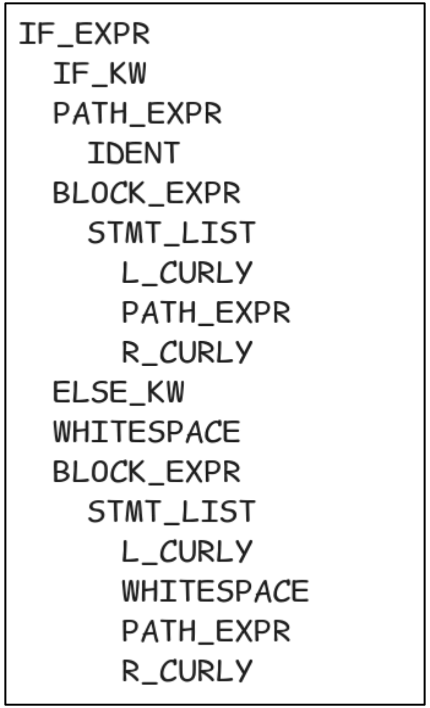
Rust CFG

How to design a Rust CFG from the Rust AST without making remaking the compiler?

But what about branching instructions?



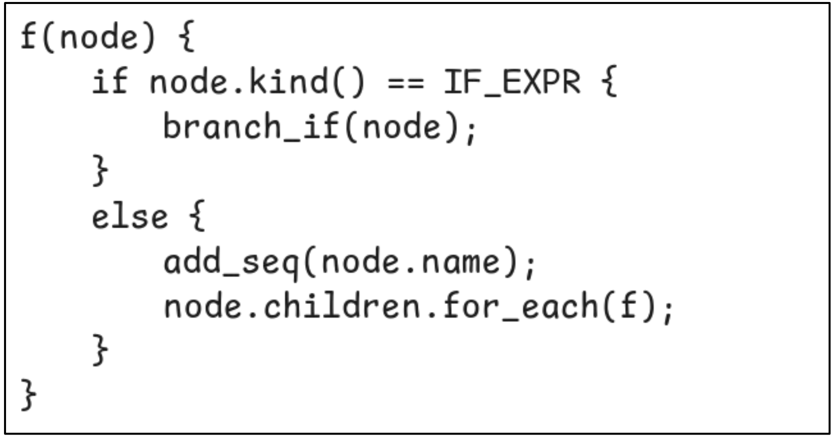
```
if cond { d1 } else { d2 }
```



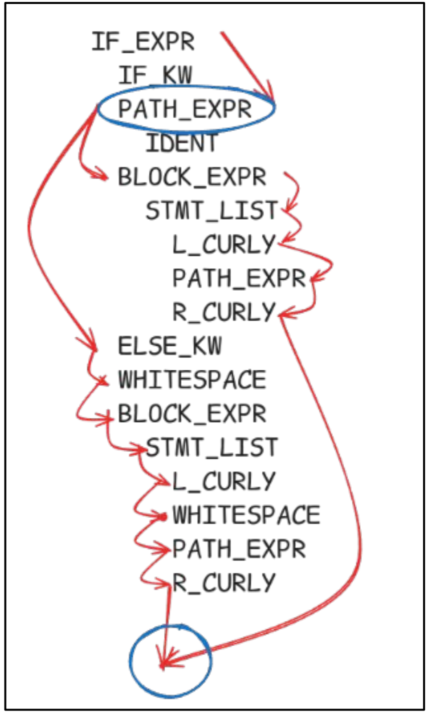
```
IF_EXPR
  IF_KW
  PATH_EXPR
    IDENT
  BLOCK_EXPR
    STMT_LIST
      L_CURLY
      PATH_EXPR
      R_CURLY
  ELSE_KW
  WHITESPACE
  BLOCK_EXPR
    STMT_LIST
      L_CURLY
      WHITESPACE
      PATH_EXPR
      R_CURLY
```

Rust CFG

How to represent rust CFG from the Rust AST without making remaking the compiler?

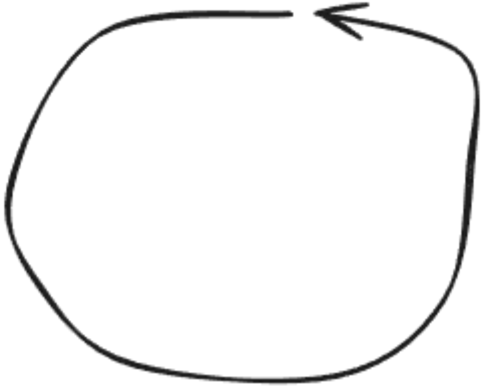


```
f(node) {
    if node.kind() == IF_EXPR {
        branch_if(node);
    }
    else {
        add_seq(node.name);
        node.children.for_each(f);
    }
}
```

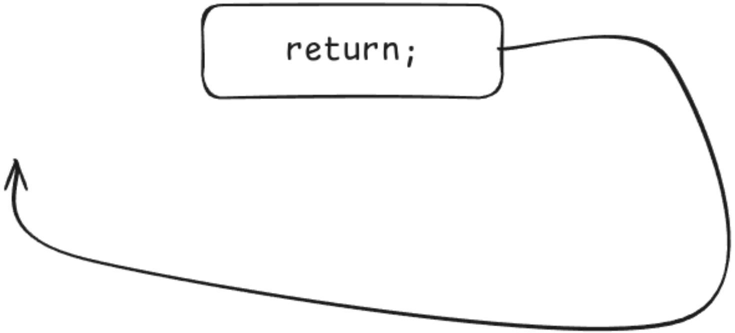


Rust CFG

Similarly we can define CFGs for loops and return statements.



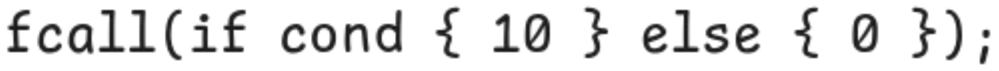
Loops



Jump statements

Problems with this approach

1. HUGE Control Flow Graphs.
2. All the CFGs shown in the slides are highly compressed. This is how a CFG looks for

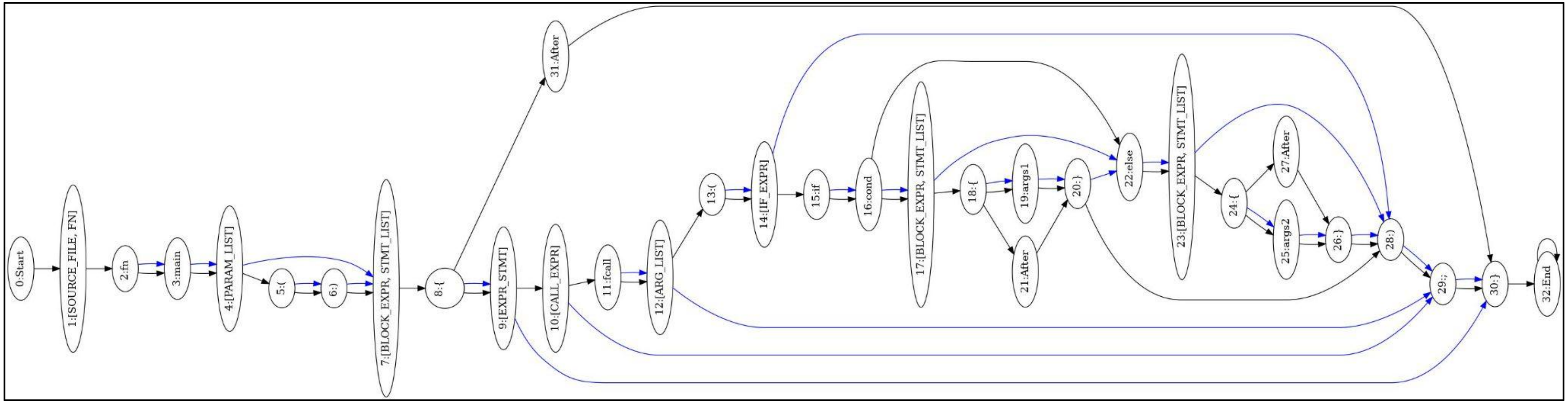


```
fcall(if cond { 10 } else { 0 });
```

Problems with this approach

1. HUGE Control Flow Graphs.
2. All the CFGs shown in the slides are highly compressed. This is how a CFG looks for

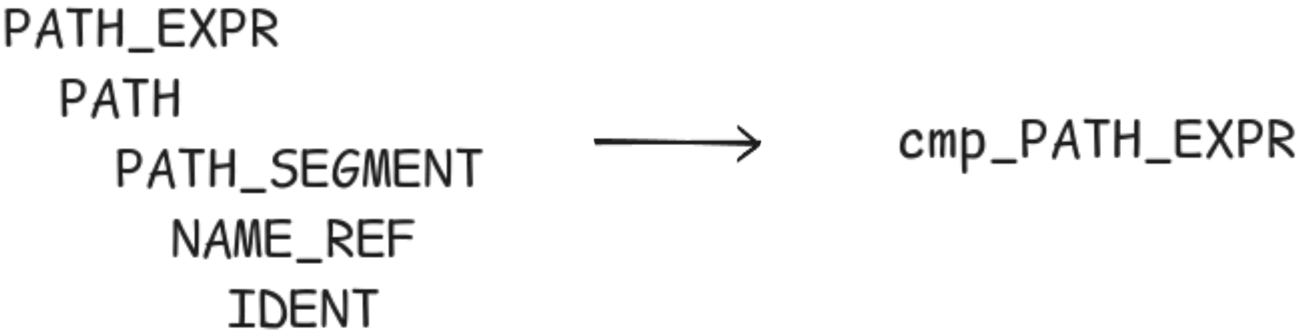
```
fcall(if cond { 10 } else { 0 });
```



Problems with this approach

1. HUGE Control Flow Graphs.

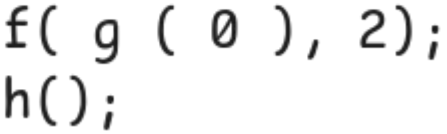
Solution :- Compress nodes with nodes with only one child.



Problems with this approach

2. Representation of metavariables

Special edges for metavariables and blocks.



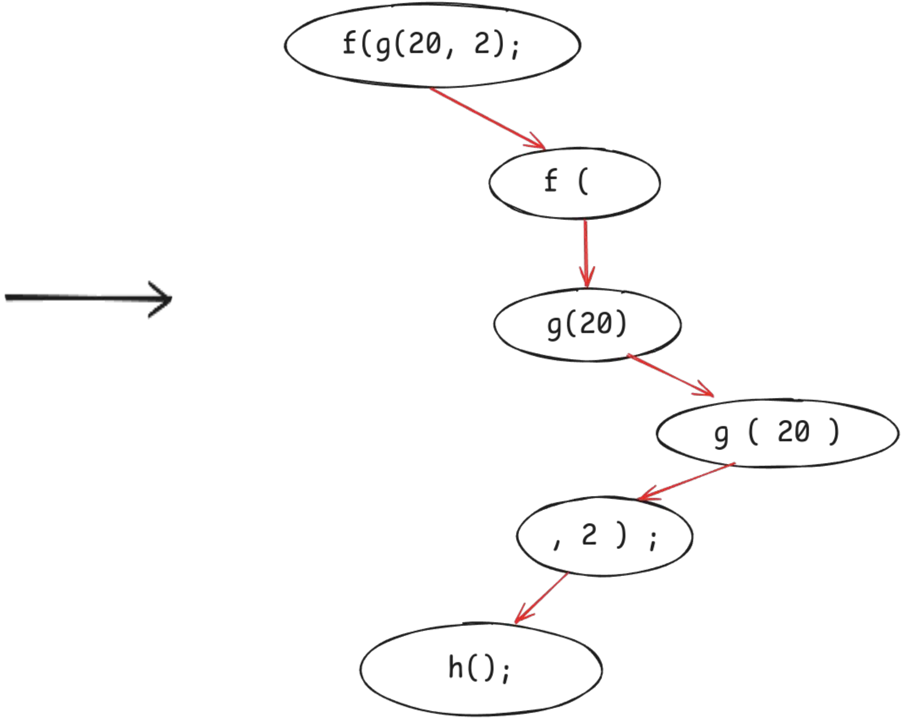
```
f( g ( 0 ), 2);
h();
```

Problems with this approach

2. Representation of metavariables

Special edges for metavariables and blocks.

```
f( g ( 0 ), 2);
h();
```



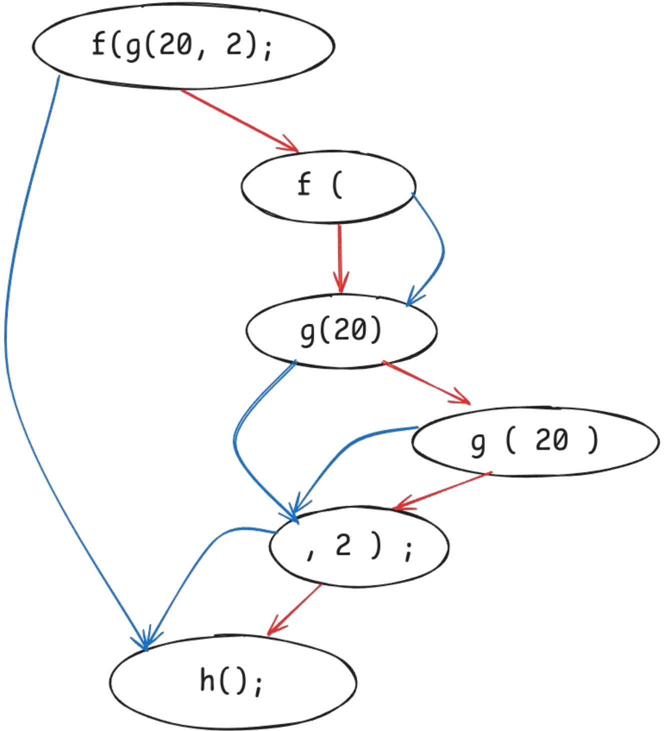
Problems with this approach

2. Representation of metavariables

Special edges (sibling) for metavariables and blocks.

```
f( g ( 0 ), 2);
h();
```

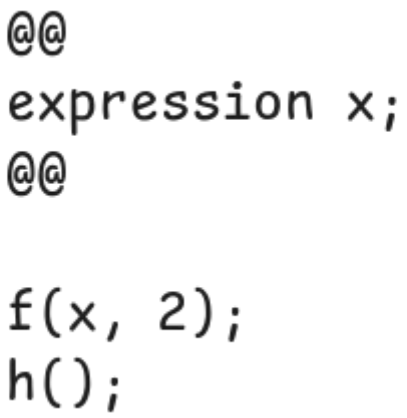
→



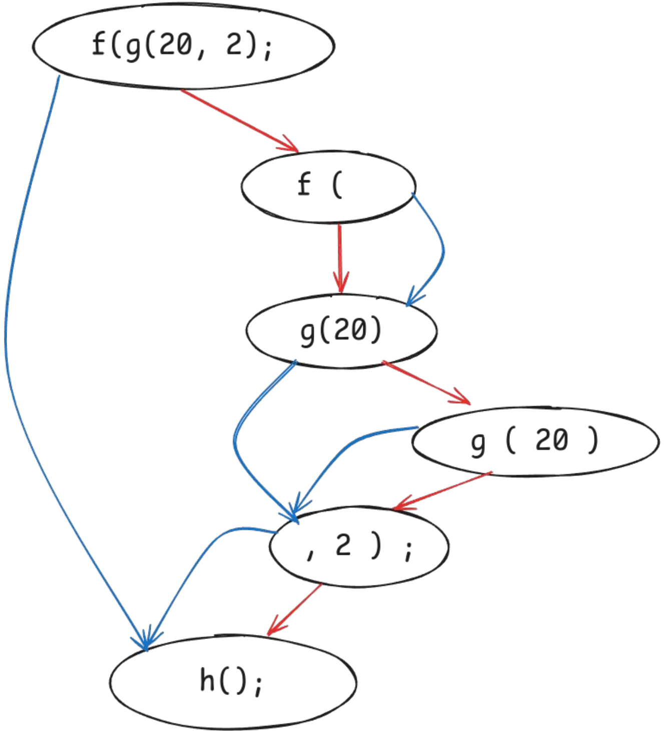
Problems with this approach

2. Representation of metavariables

Special edges (sibling) for metavariables and blocks.



```
@@
expression x;
@@

f(x, 2);
h();
```



Other points

CTL formulas are very verbose and hard to read in their current state. For example…

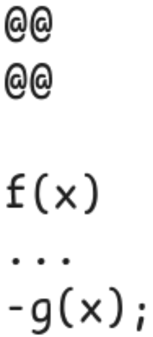
Other points

CTL formulas are very verbose and hard to read in their current state

```
[EXPR_STMT]  & (AX ([CALL_EXPR]  & (AX (f  & (AX ([ARG_LIST]  & (AX (Exnk l1 ((
& (Paren(l1)) & (AX (Ex x (x  & (AX ()(M)  & (Paren(l1)) & (AX (;  & (AX (
A[NOT ([EXPR_STMT]  & (AX ([CALL_EXPR]  & (AX (f  & (AX ([ARG_LIST]  & (AX
(Exnk l1 ((  & (Paren(l1)) & (AX (Ex x (x  & (AX ()(M)  & (Paren(l1)) & (AX (; ))))) OR
After))))))))))) OR [EXPR_STMT]  & (AX ([CALL_EXPR]  & (AX (Ex _v (g ) & (AX
([ARG_LIST]  & (AX (Exnk l1 (Ex _v (( ) & (Paren(l1)) & (AX (Ex _v (x ) & (AX (Ex _v
()(M) ) & (Paren(l1)) & (AX (Ex _v (; ))))) OR After)))))))))))) U [EXPR_STMT]  & (AX
([CALL_EXPR]  & (AX (Ex _v (g modif) & (AX ([ARG_LIST]  & (AX (Exnk l1 (Ex _v ((
modif) & (Paren(l1)) & (AX (Ex _v (x modif) & (AX (Ex _v ()(M) modif) & (Paren(l1)) &
(AX (Ex _v (; modif))))) OR After)))))))))))]))))))) OR After)))))))))))
```

Other points

CTL formulas are very verbose and hard to read in their current state



```
@@
@@

f(x)
...
-g(x);
```

Thankfully the CfR user does not have to deal with CTL formulas :)

Latest Developments

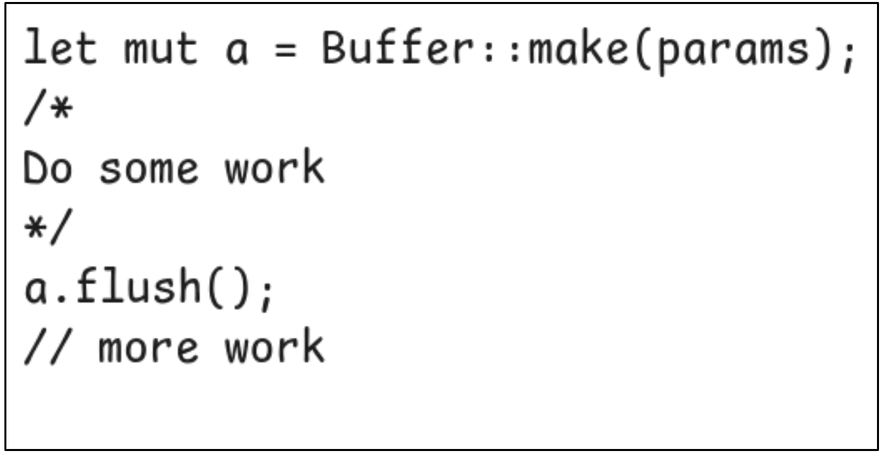
Ellipses (...)

a. The ellipses operator
b. Matches any control flow path connecting two nodes
c. Helpful for when we don’t care about intermediate statements
d. Finds all paths by default

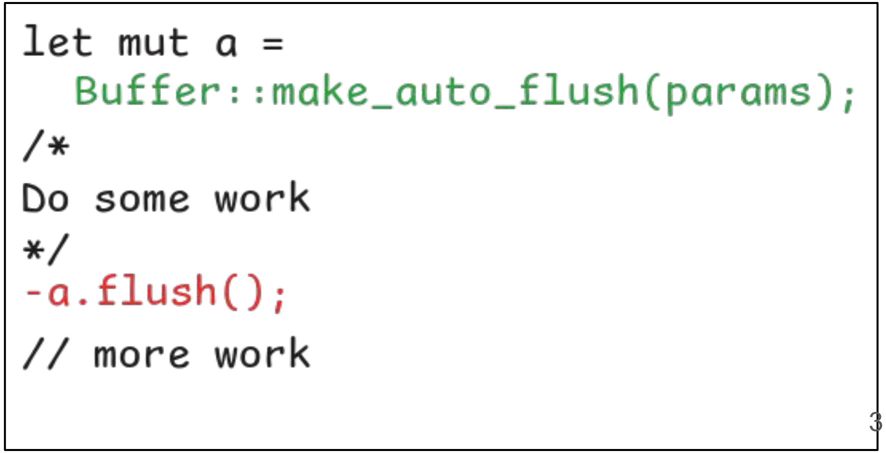
Latest Developments

Ellipses (…)

a. The ellipses operator
b. Matches any control flow path connecting two nodes
c. Helpful for when we don’t care about intermediate statements
d. Finds all paths by default



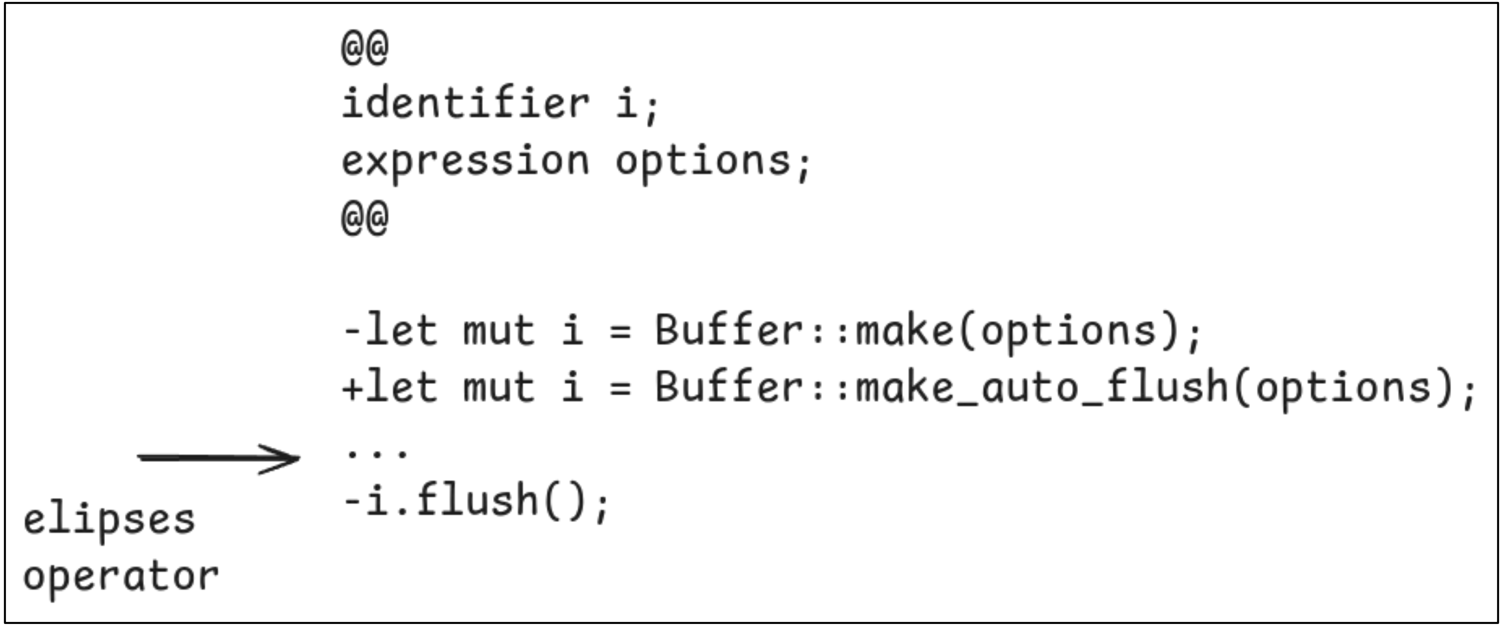
```
let mut a = Buffer::make(params);
/*
Do some work
*/
a.flush();
// more work
```



```
let mut a =
  Buffer::make_auto_flush(params);
/*
Do some work
*/
-a.flush();
// more work
```

Latest Developments

Ellipses (…)



```
@@
identifier i;
expression options;
@@

-let mut i = Buffer::make(options);
+let mut i = Buffer::make_auto_flush(options);
...
-i.flush();
```

elipses
operator

Latest Developments

Disjunctions

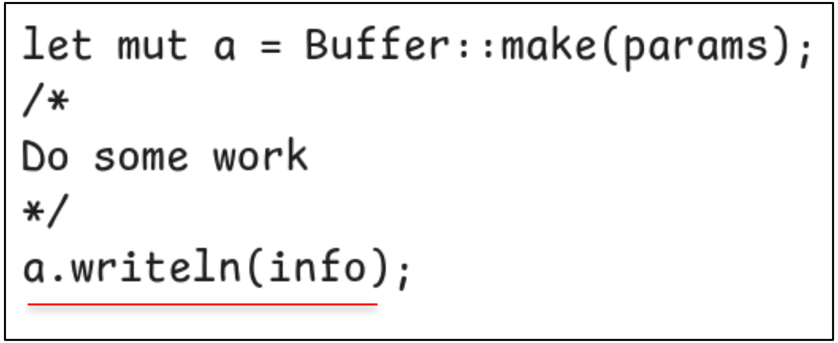
1. Conditional Matching
2. Matches either one of the branches

Disjunctions

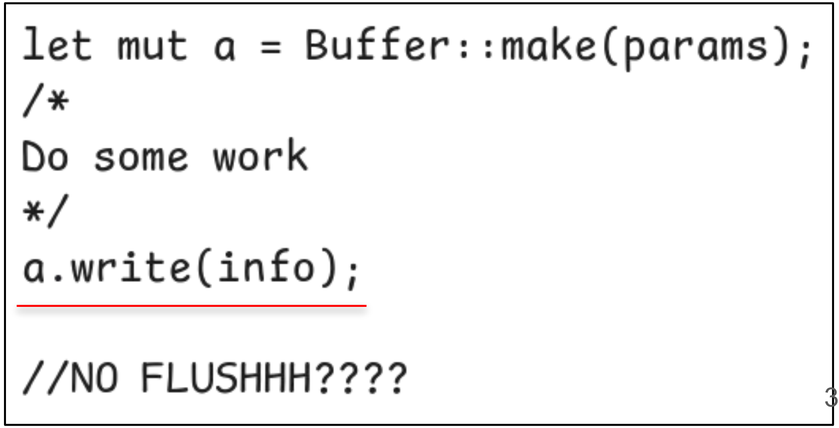
Example

write() needs to be flushed always,

writeln() is self flushing.

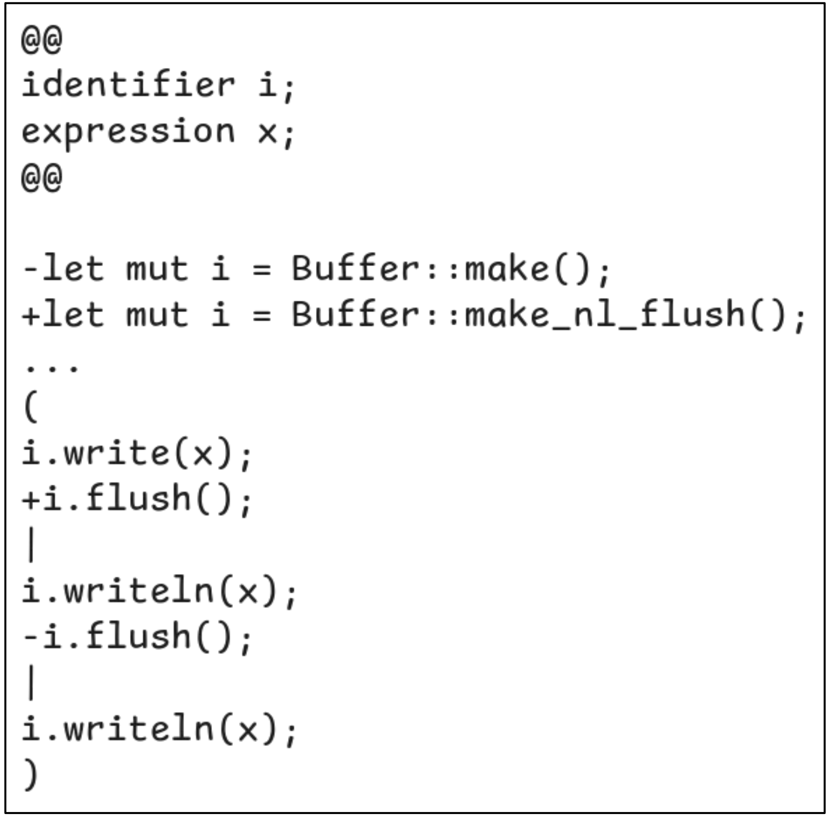


```
let mut a = Buffer::make(params);
/*
Do some work
*/
a.writeln(info);
```

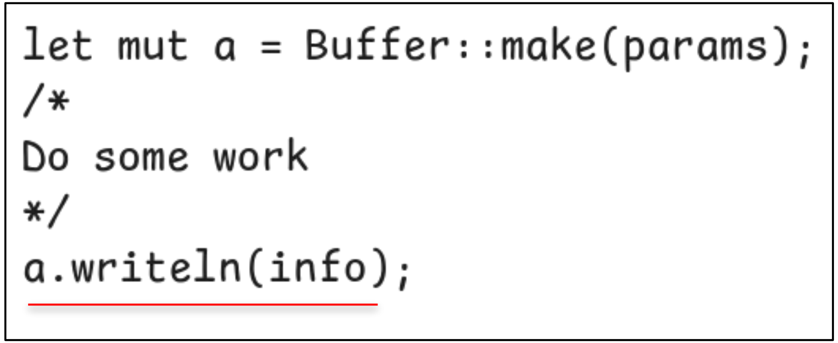


```
let mut a = Buffer::make(params);
/*
Do some work
*/
a.write(info);

//NO FLUSHHH????
```

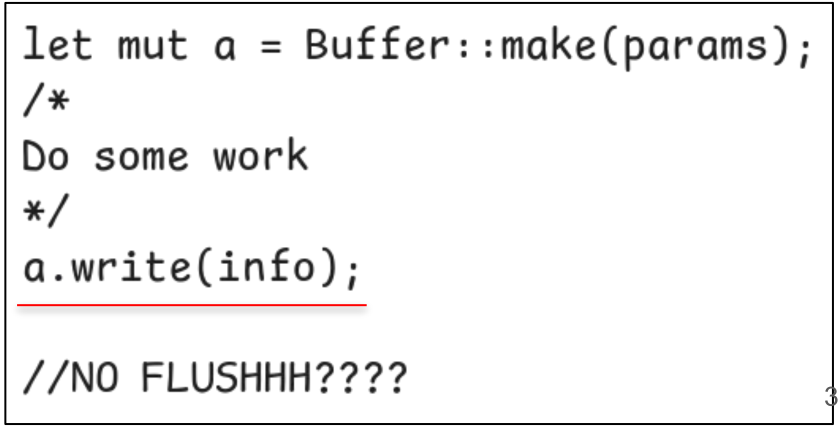
Disjunctions



```
@@
identifier i;
expression x;
@@

-let mut i = Buffer::make();
+let mut i = Buffer::make_nl_flush();
...
(
i.write(x);
+i.flush();
|
i.writeln(x);
-i.flush();
|
i.writeln(x);
)
```



```
let mut a = Buffer::make(params);
/*
Do some work
*/
a.writeln(info);
```



```
let mut a = Buffer::make(params);
/*
Do some work
*/
a.write(info);

//NO FLUSHHH????
```

Disjunctions

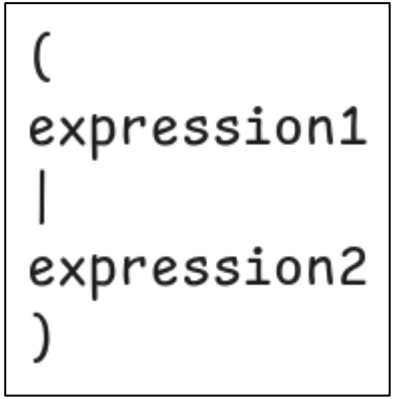
WHAT’S NEW?????

Disjunctions

Disjunction branches can now be anything as long as the whole patch makes sense.

Disjunctions

Disjunction branches can now be anything as long as the whole patch makes sense.

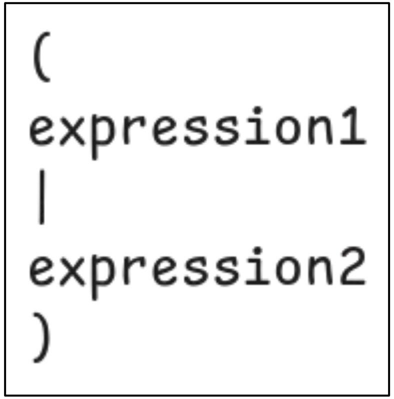


```
(
expression1
|
expression2
)
```

Previously

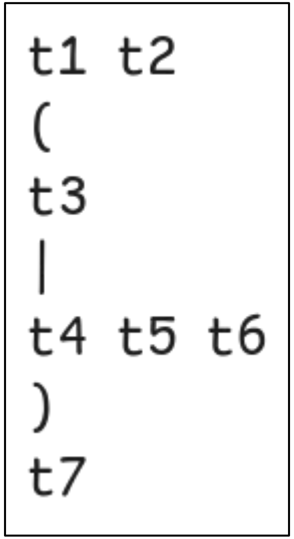
Disjunctions

Disjunction branches can now be anything as long as the whole patch makes sense.

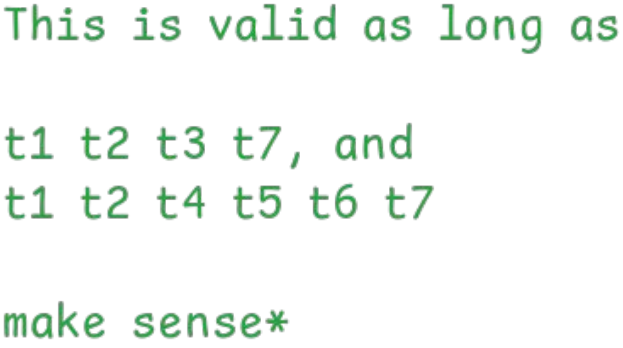


```
(
expression1
|
expression2
)
```

Previously



```
t1 t2
(
t3
|
t4 t5 t6
)
t7
```



```
This is valid as long as

t1 t2 t3 t7, and
t1 t2 t4 t5 t6 t7

make sense*
```

* For the most part

Disjunctions

A bit of background

1. Coccinelle For C uses its own handwritten parser for both C and SmPL.
2. Coccinelle For Rust uses RustAnalyzer for **both**.
3. Using RA has its upsides and downsides, but it's mostly positive.

Disjunctions

We did not want to extend the rust parser with SmPL constructs because of maintainability and time constraints.

Disjunctions

We did not want to extend the rust parser with SmPL constructs because of maintainability and time constraints.

Solution:- Convert all SmPL constructs into Rust parsable syntax.

Disjunctions

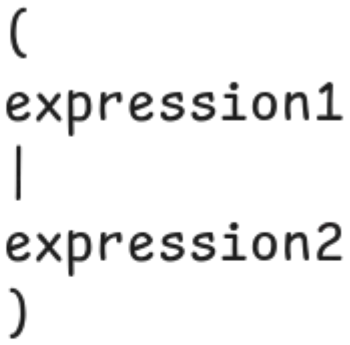
Previously on Coccinelle For Rust...

1. Disjunctions -> If statements

Disjunctions

Previously on Coccinelle For Rust...

1. Disjunctions -> If statements

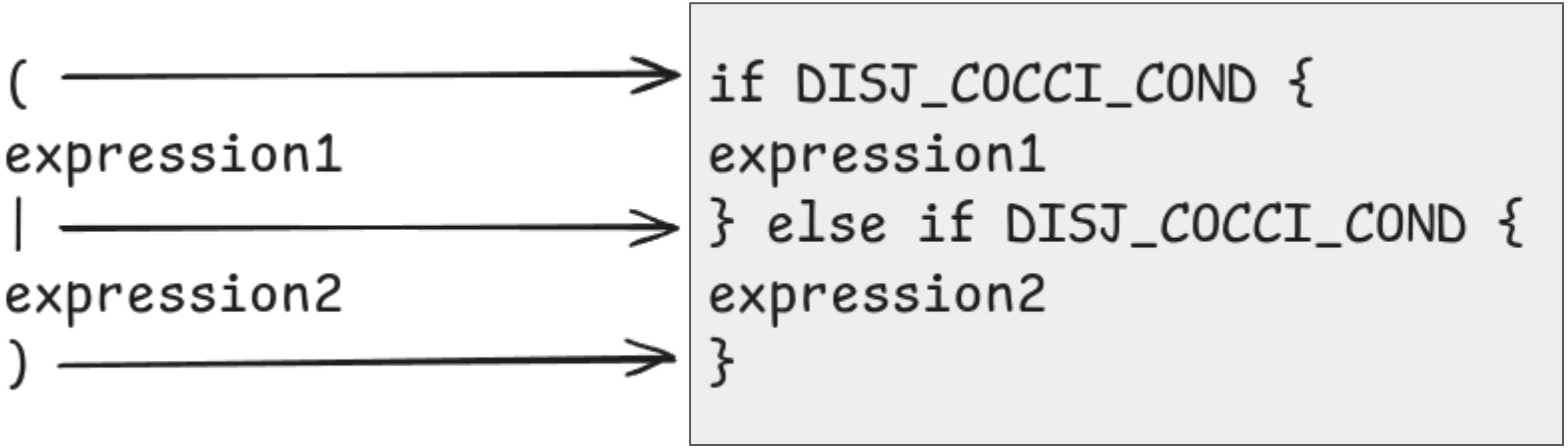


```
(
expression1
|
expression2
)
```

Disjunctions

Previously on Coccinelle For Rust...

1. Disjunctions -> If statements

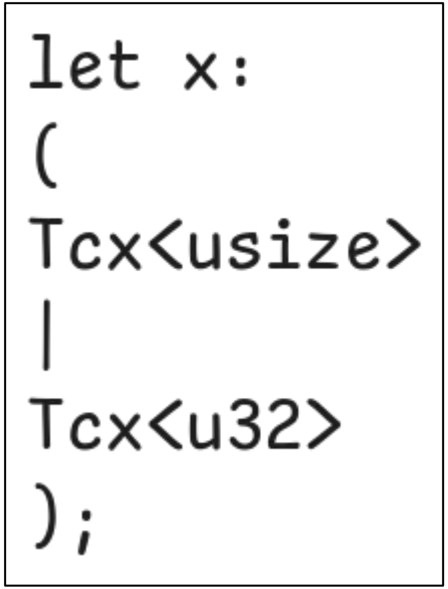


```
(
expression1
|
expression2
)
```

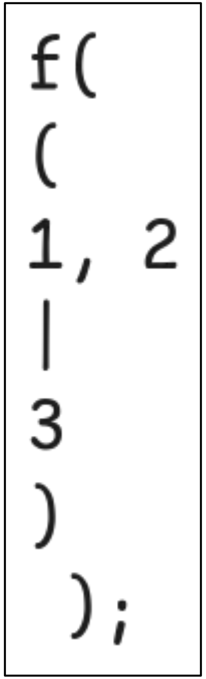
```
if DISJ_COCCI_COND {
expression1
} else if DISJ_COCCI_COND {
expression2
}
```

Disjunctions

Non-expression disjunctions



```
let x:
(
Tcx<usize>
|
Tcx<u32>
);
```



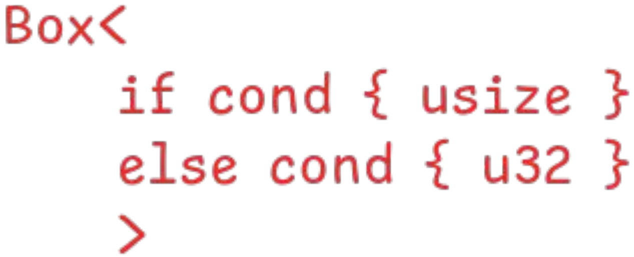
```
f(
(
1, 2
|
3
)
 );
```

Disjunctions

Previously on Coccinelle For Rust…

1. Disjunctions -> If statements

Problem: Cannot parse anything other than expressions.



```
Box<
    if cond { usize }
    else cond { u32 }
    >
```

✗

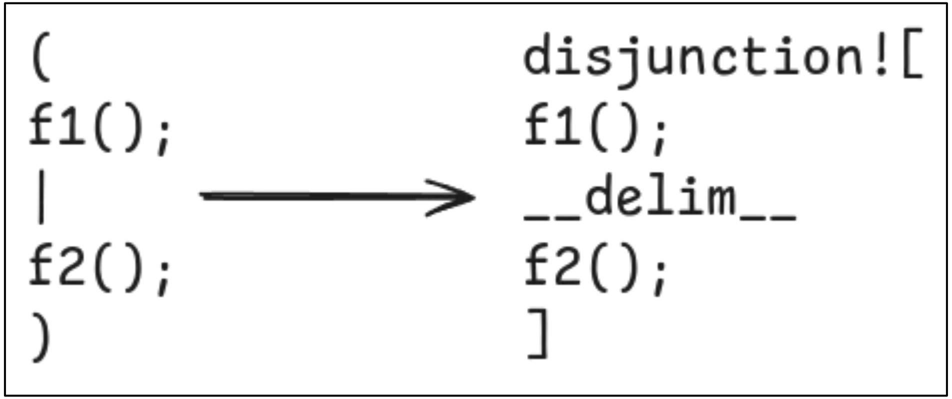
Disjunctions

HAIL OUR SAVIOUR :- ***MACROS***

1. Rust macros are very versatile. There are a few types of declarative rust macros :-
   a. MacroDef
   b. MacroCall
   c. MacroType
   d. …?
2. If we use macros to wrap our disjunctions, not only can we parse them, but also get what should be in their place.

Disjunctions

What we do now :-



```
(                        disjunction![
f1();                    f1();
|          →             __delim__
f2();                    f2();
)                        ]
```

Disjunctions

What we do now :-

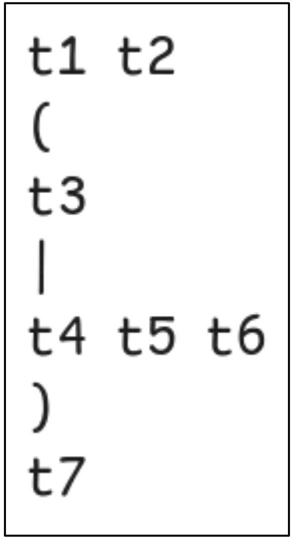
1. We can have as many nested disjunctions as we want

2. All the nested macros are automatically parsed and grouped

3. Accounts for more situations

Disjunctions

What we do now :-



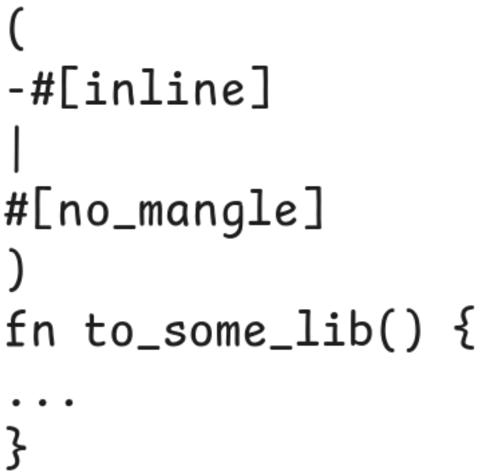
```
t1 t2
(
t3
|
t4 t5 t6
)
t7
```

1. Parse disjunctions
2. Get all possible paths as a string but keep the disjunction information
3. Make sure that these branches are parsable
4. Parse the newly formed branches
5. Merge all the branches into one disjunction

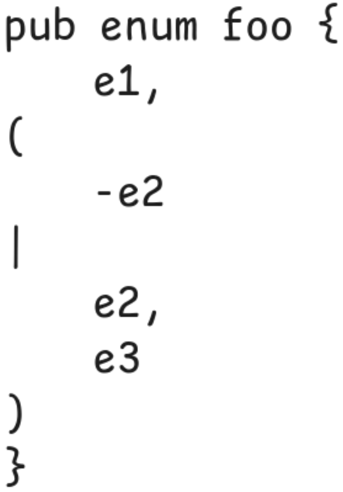
Disjunctions

Note:-

There are still cases where disjunctions cannot be used. For example :-



```
(
-#[inline]
|
#[no_mangle]
)
fn to_some_lib() {
...
}
```

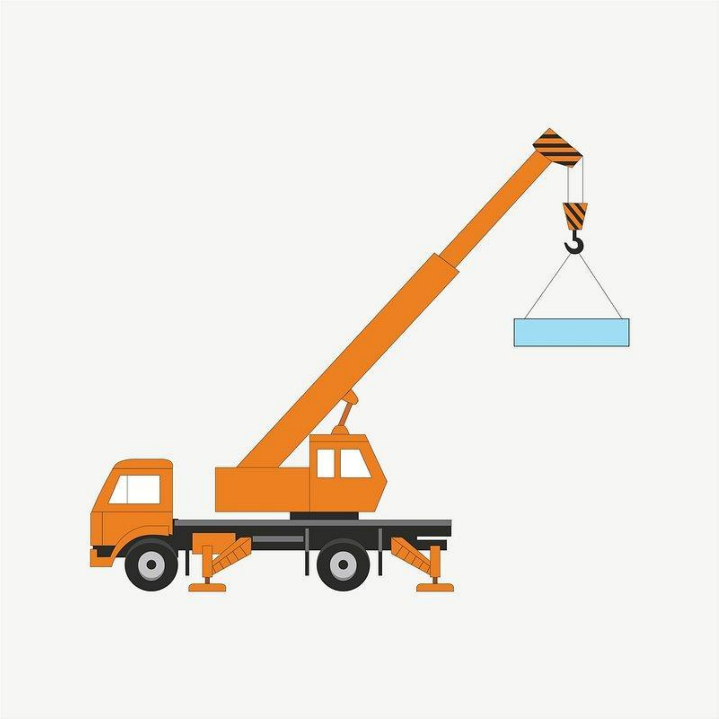


```
pub enum foo {
    e1,
(
    -e2
|
    e2,
    e3
)
}
```

Latest Developments

Disjunctions

Still a work in progress :)



Remaining Challenges

Macros

1. They are a pain in the AST
2. CfR uses rustfmt
3. rustfmt does not format macros and mods properly
4. Ambiguity as to what to do.

Parallelization

1. Limited parallelization capabilities due to the thread-unsafe structure of rowan syntax nodes.

Future Plans

1. Completing disjunction integration
2. Add more modifiers (* ?)
3. Better parallelization
4. Filter files by constants
5. more…

Possible applications?

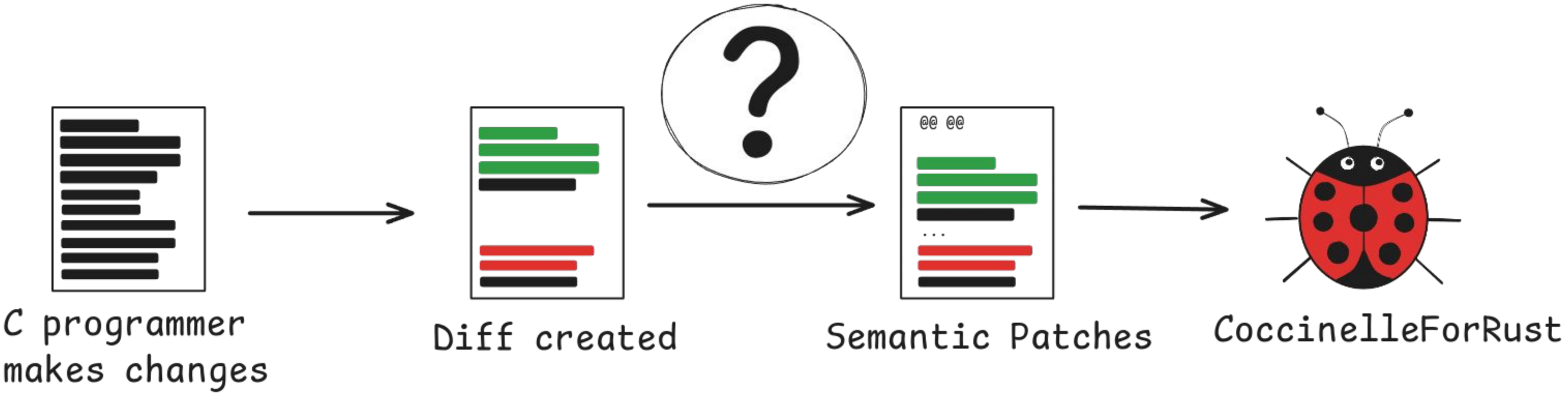
Interfacing C-Rust Code

a. Changes in C side of the code require the corresponding Rust code to be updated.

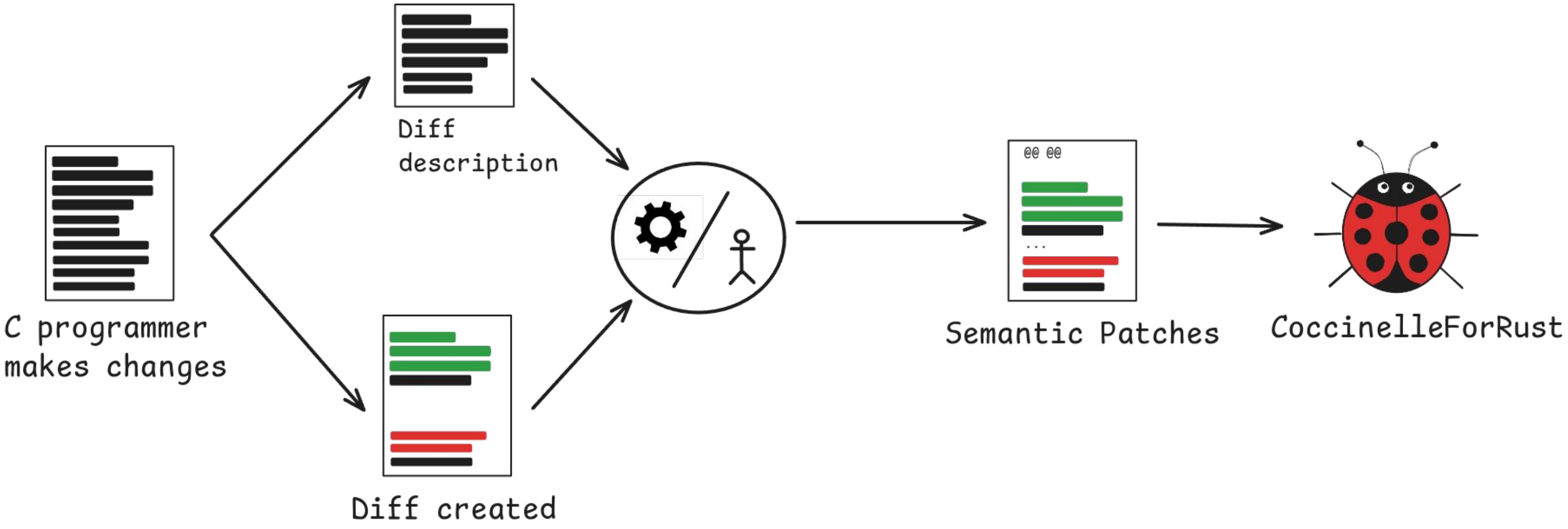
b. Questions in the community as to who should make the changes across languages.

Coccinelle For Rust could potentially act as a tool which automates the changes from C to Rust. This reduces the burden on both C and Rust developers.

Possible applications?



Possible applications?



Possible applications?

Some examples of the diff description -

1. arg2 is never NULL
2. struct foo *arg1 can only be dereferenced once
3. Size of `struct foo` has changed to 32 bytes from 48 bytes.
4. More...

Question to the audience

What kind of C-Rust interface changes are most common in the linux kernel and would benefit most from automation?

Support

COLLABORA

Thank you Collabora for supporting the development of Coccinelle For Rust!

COCCINELLE FOR RUST LINKS

1. Main Page - https://rust-for-linux.com/coccinelle-for-rust
2. Gitlab Page - https://gitlab.inria.fr/coccinelle/coccinelleforrust/-/tree/main?ref_type=heads (Please use the ctl2 branch, as per the link)
3. Previous Talks - https://gitlab.inria.fr/coccinelle/coccinelleforrust/-/blob/ctl2/talks/lpc23.pdf?ref_type=heads
4. Contact: julia.lawall@inria.fr

   tathagata.roy1278@gmail.com