

# Clone Detection and Removal for Erlang/OTP within a Refactoring Environment

Huiqing Li  
Simon Thompson

Computing Lab, University of Kent, UK

# Overview

Refactoring and duplicate code detection.

Erlang/OTP and Wrangler.

The Wrangler Clone Detector.

Refactoring Support for Clone Removal

Case Studies

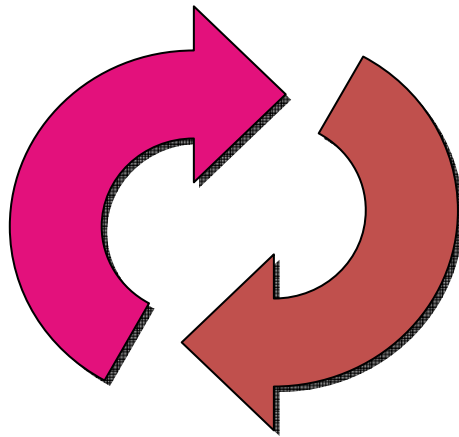
Related Work

Conclusions and Future Work

# Refactoring

Refactoring means changing the **design** or **structure** of a program ... without changing its **behaviour**.

Modify



Refactor

# Refactoring tool support

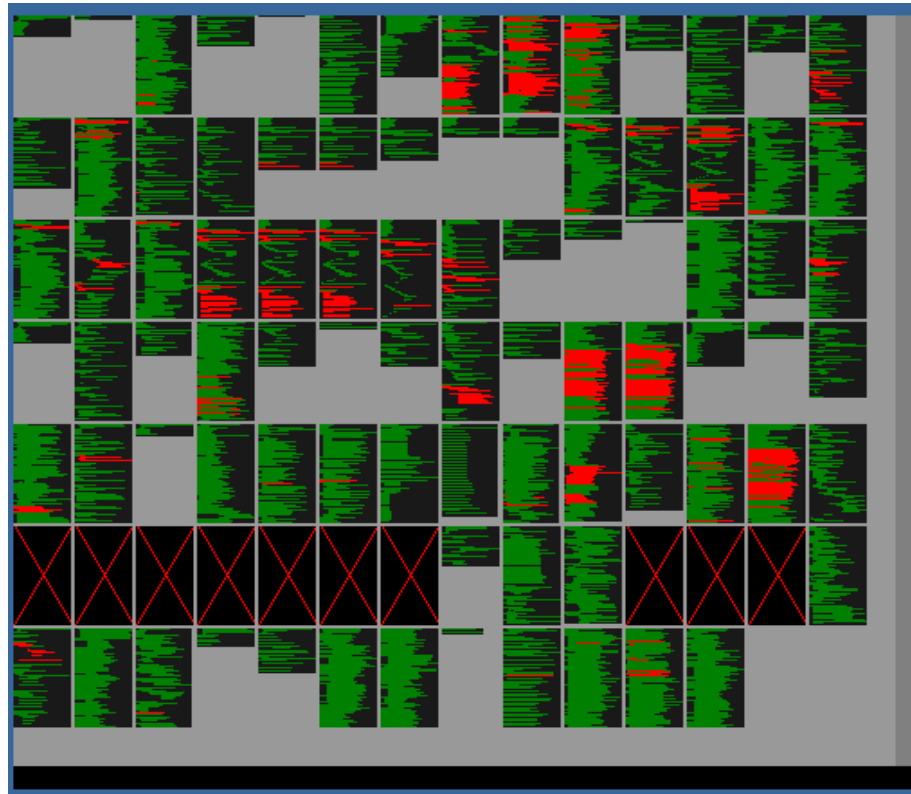
Bureaucratic and  
diffuse.

Tedious and error  
prone.

Semantics: scopes,  
types, modules, ...

Undo/redo

Enhanced creativity



# Code Smells

## Bad code smells

- Name does not reflect the meaning
- Function too long
- Code not actually used
- Bad module structure
- Excessive nesting
- ***Duplicated code***
- .....

# Duplicate Code Detection

- Duplicated code, or code cloning, refers to a program fragment that is **identical** or **similar** to another.
- The prime reason of code duplication is reusing of code, logic, or design.
- Program design problem: lack of encapsulation or abstraction.

# Duplicate Code Considered Harmful

- Why?
  - Increases the probability of bug propagation.
  - Increases the size of the source code and the executable.
  - Increases compile time.
  - Increases the cost of maintenance.
- But not always harmful ... ?

# Erlang/OTP



Erlang: functional language with built-in support for concurrency, distribution and fault-tolerance.

OTP : a middleware platform & design principles.

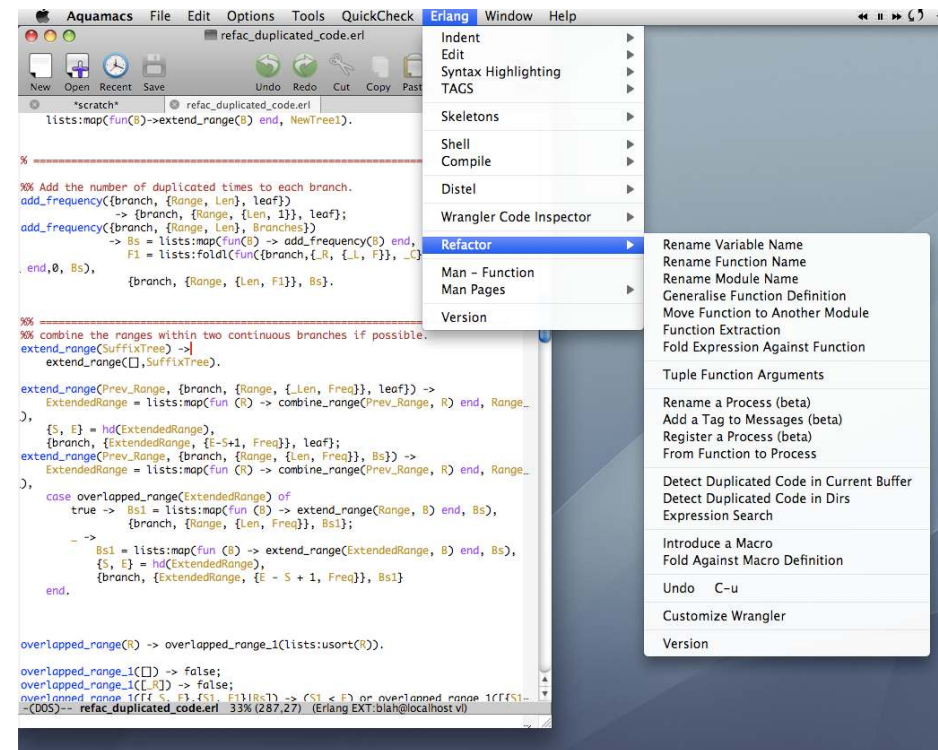
```
%% Factorial in Erlang.  
-module (fac).  
  
-export ([fac/1]).  
  
fac(0) -> 1;  
fac(N) when N > 0 -> N * fac(N-1).
```

Some eccentricities: multiple binding occurrences, bound variables in patterns, side-effects ... .

# Wrangler

Refactoring tool for Erlang, embedded in emacs and (soon) in Eclipse.

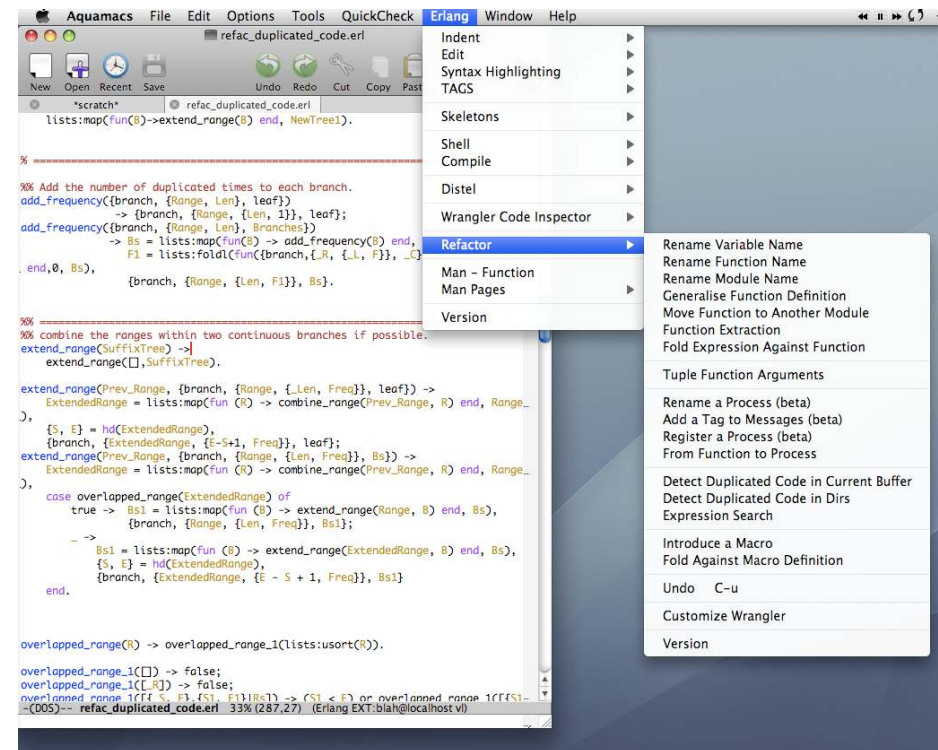
- Structural: renaming, generalisation, tupling, ...
- Modules: rename, move functions, batch, ...
- Processes: name a process, tag messages, ...
- Macro sensitive



# Wrangler

Refactoring tool for Erlang, embedded in emacs and (soon) in Eclipse.

- Supports undo of refactorings.
- Multi-module refactorings.
- Bad smell detection.



# The Wrangler Clone Detector

Especially for Erlang/OTP programs.

Makes use of both syntactic and static semantic information.

Integrated within the refactoring environment.

Reports syntactically well-formed code fragments that are similar ...

# Similar code fragments

Syntactically well-formed code fragments that are identical after consistent renaming of variables, with variations in literals, layout and comments.

# The Wrangler Clone Detector

- Common terminology.
  - Clone pair: a pair of code fragments that are identical or similar to each other.
  - Clone class: a maximal set of code fragments in which any two of them form a clone pair.
- Threshold values.
  - The minimum number of tokens for a code clone.
  - The minimum number of members of a clone class.

# The Wrangler Clone Detector

- The Wrangler clone detector makes use of both token stream and AST annotated with static semantic information.
  - Token-based approaches
    - Efficient.
    - Report non-syntactic clones.

```
-> case lists:subtract(SLocs1, ELocs2) of
  []-> R1= lists:filter(fun({S, E}) -> lists:member(E, SLocs1) end, Range),
      R2= lists:map(fun({S, E}) -> S end, R1),
      {lists:zip(R2, ELocs1), Len1+Len2, F1};
  _ ->
```

(a)

```
-> case lists:subtract(ELocs1, SLocs2) of
  [] -> R3= lists:filter(fun({S,E}) -> lists:member(S, ELocs1) end, Range),
      R4 = lists:map(fun({S,E}) -> E end, R3),
      {lists:zip(SLocs1, R4), Len1+Len2, F1};
  _ ->
```

(b)

# The Wrangler Clone Detector

- The Wrangler clone detector makes use of both token stream and AST annotated with static semantic information.
  - Token-based approaches
    - Efficient.
    - Report non-syntactic clones.

```
-> case lists:subtract(SLocs1, ELocs2) of
  []-> R1= lists:filter(fun({S, E}) -> lists:member(E, SLocs1) end, Range),
        R2= lists:map(fun({S, E}) -> S end, R1),
        {lists:zip(R2, ELocs1), Len1+Len2, F1};
  _ ->
```

(a)

```
-> case lists:subtract(ELocs1, SLocs2) of
  [] -> R3= lists:filter(fun({S,E}) -> lists:member(S, ELocs1) end, Range),
        R4 = lists:map(fun({S,E}) -> E end, R3),
        {lists:zip(SLocs1, R4), Len1+Len2, F1};
  _ ->
```

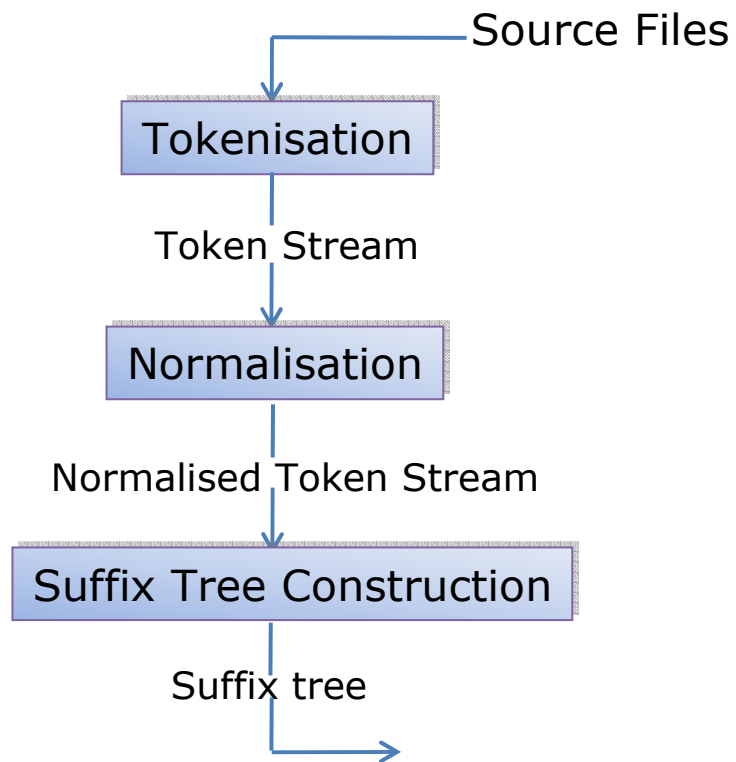
(b)

# The Wrangler Clone Detector

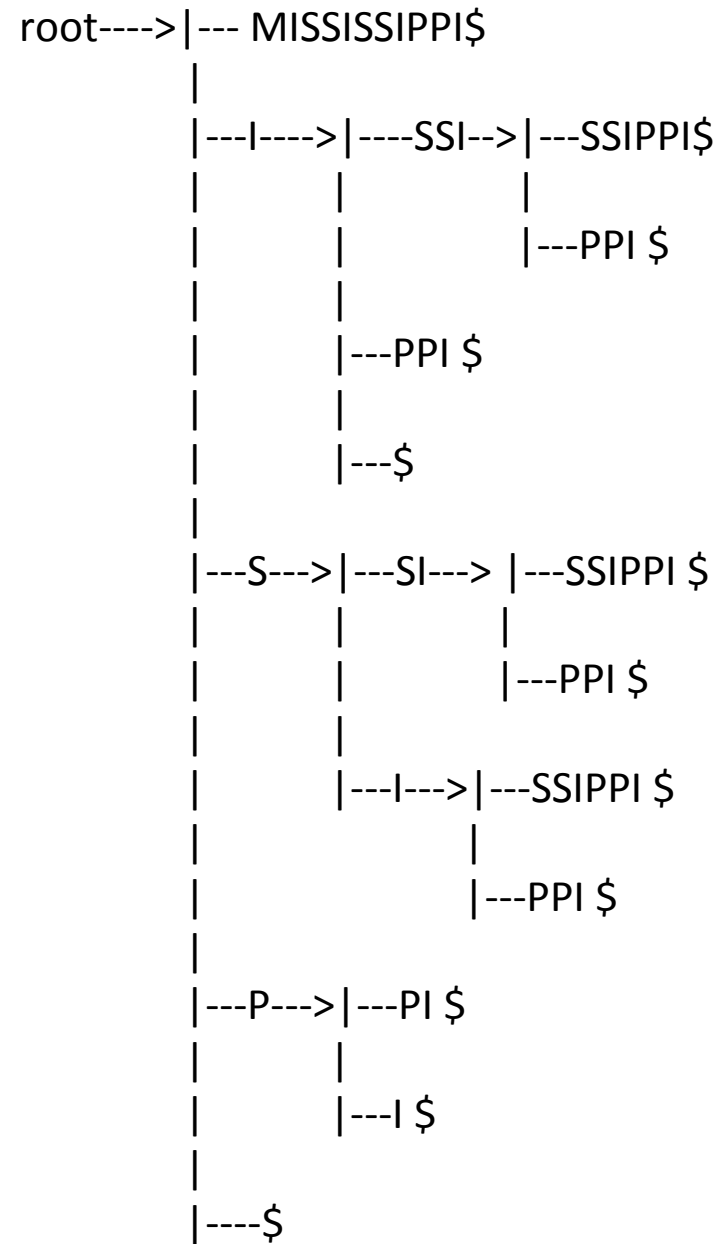
- The Wrangler clone detector makes use of both token stream and AST annotated with static semantic information.
  - Token-based approaches
    - Efficient.
    - Report non-syntactic clones.
    - Not very suitable for consistent renaming checking for languages that allow nested scopes.
  - AST-based approaches
    - Less efficient.
    - Report syntactic clones.
    - Checking for consistent renaming is easier.

# The Wrangler Clone Detector

- An overview of the Wrangler approach

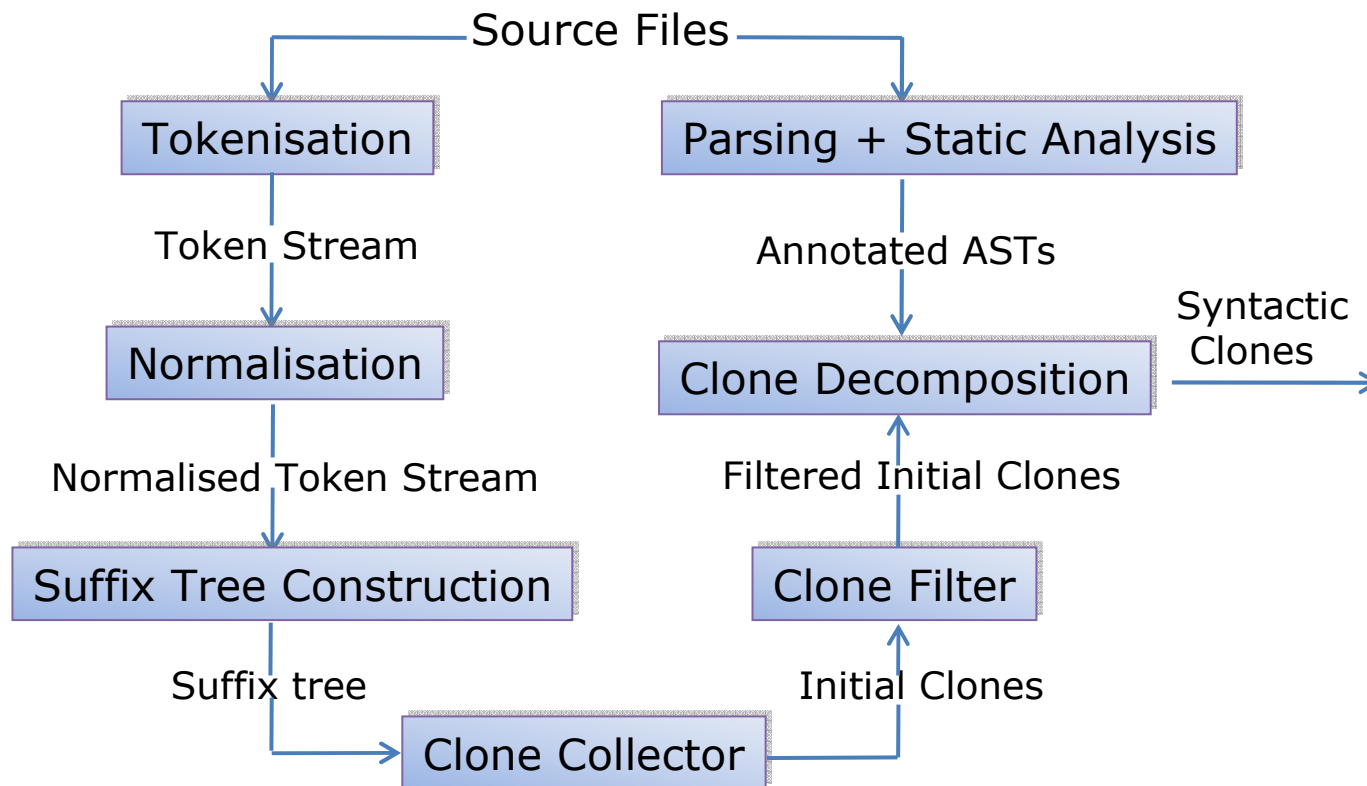


# Suffix tree for Mississippi



# The Wrangler Clone Detector

- An overview of the approach



# The Wrangler Clone Detector

- Decomposition of clones into syntactic units.

```
-> case lists:subtract(SLocs1, ELocs2) of
  []-> R1= lists:filter(fun({S, E}) -> lists:member(E, SLocs1) end, Range),
      R2= lists:map(fun({S, E}) -> S end, R1),
      {lists:zip(R2, ELocs1), Len1+Len2, F1};
  _ ->
```

(a)

```
-> case lists:subtract(ELocs1, SLocs2) of
  [] -> R3= lists:filter(fun({S,E}) -> lists:member(S, ELocs1) end, Range),
      R4 = lists:map(fun({S,E}) -> E end, R3),
      {lists:zip(SLocs1, R4), Len1+Len2, F1};
  _ ->
```

(b)

# The Wrangler Clone Detector

- Decomposition of clones into syntactic units.

```
lists:subtract(SLocs1, ELocs2) (a)
```

```
lists:subtract(ELocs1, SLocs2) (b)
```

Clone pair 1

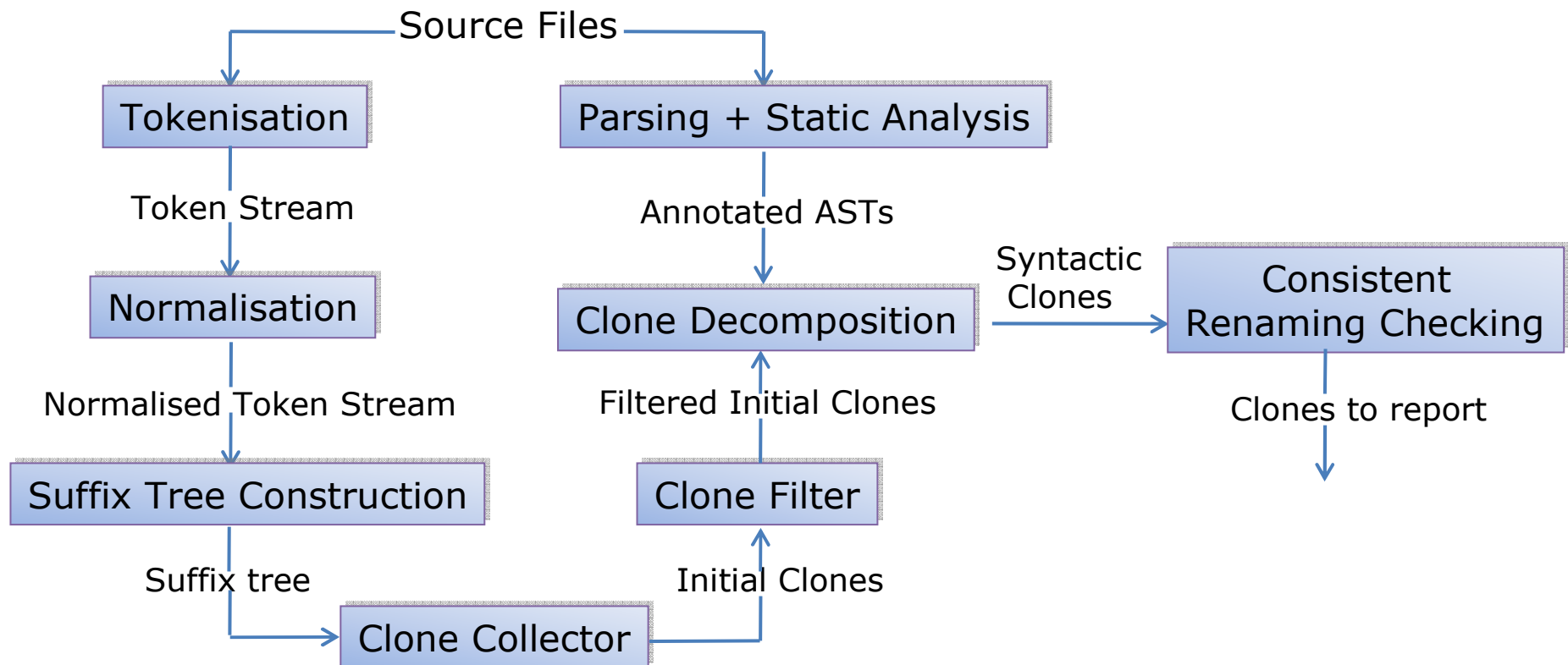
```
R1= lists:filter(fun({S, E}) -> lists:member(E, SLocs1) end, Range),  
R2= lists:map(fun({S, E}) -> S end, R1),  
{lists:zip(R2, ELocs1), Len1+Len2, F1}; (a)
```

```
R3= lists:filter(fun({S,E}) -> lists:member(S, ELocs1) end, Range),  
R4 = lists:map(fun({S,E}) -> E end, R3),  
{lists:zip(SLocs1, R4), Len1+Len2, F1}; (b)
```

Clone pair 2

# The Wrangler Clone Detector

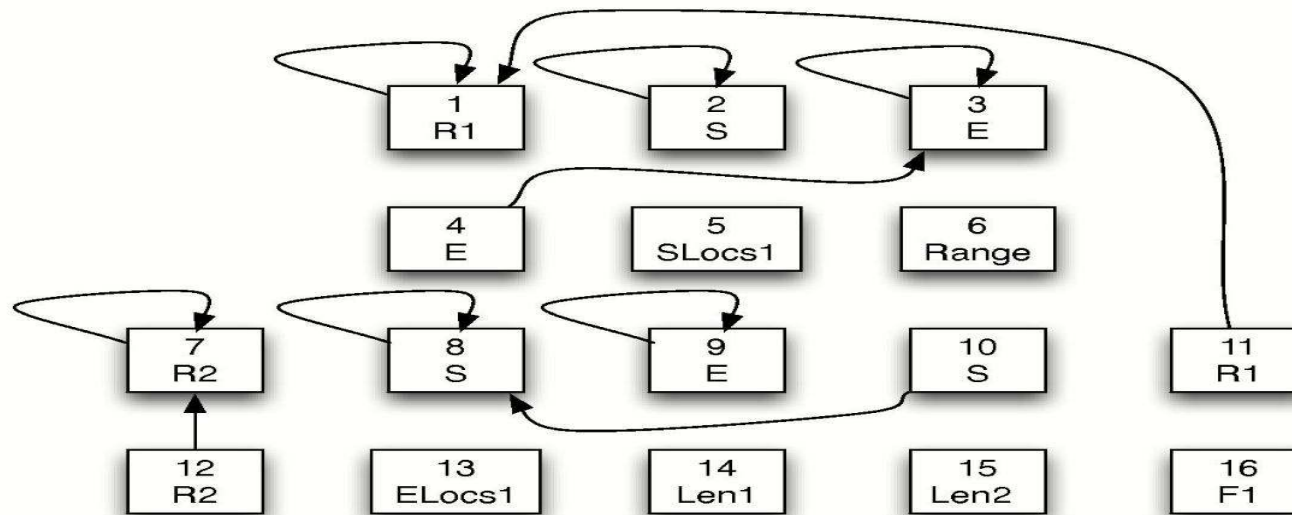
- An overview of the approach



# The Wrangler Clone Detector

- Checking for consistent renaming of variables by comparing the binding graph of member clones.

```
R1= lists:filter(fun({S, E}) -> lists:member(E, SLocs1) end, Range),  
R2= lists:map(fun({S, E}) -> S end, R1),  
{lists:zip(R2, ELocs1), Len1+Len2, F1};
```



# The Wrangler Clone Detector

- Checking for consistent renaming of variables by comparing the binding graph of member clones.

```
R1= lists:filter(fun({S, E}) -> lists:member(E, SLocs1) end, Range),  
R2= lists:map(fun({S, E}) -> S end, R1),  
{lists:zip(R2, ELocs1), Len1+Len2, F1};
```

 (a)

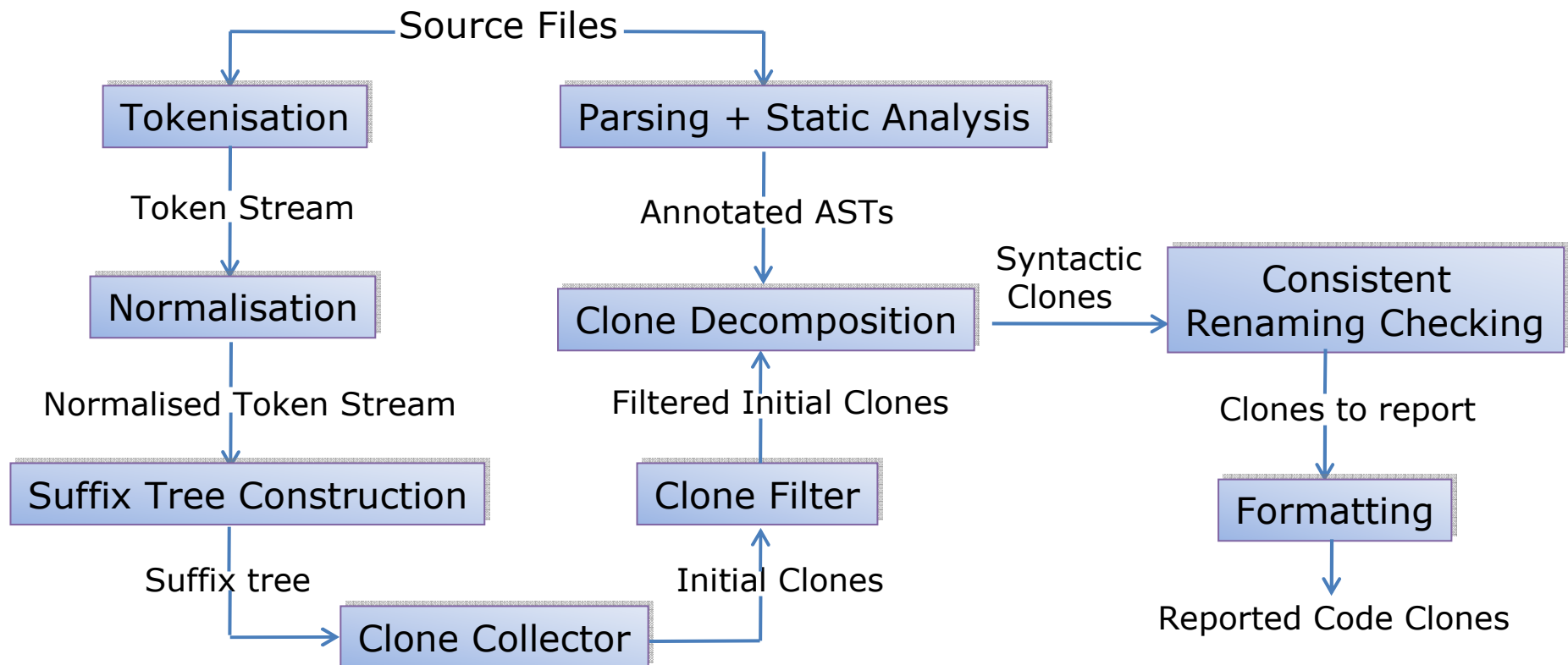
```
R3= lists:filter(fun({S,E}) -> lists:member(S, ELocs1) end, Range),  
R4 = lists:map(fun({S,E}) -> E end, R3),  
{lists:zip(SLocs1, R4), Len1+Len2, F1};
```

 (b)

Clone pair 2

# The Wrangler Clone Detector

- An overview of the approach



# The Wrangler Clone Detector

*Demo*

# Refactoring Support for Clone Removal

- Refactorings
  - Function extraction.
  - Generalise a function definition.
  - Fold against a function definition.
  - Move a function from one module to another

*Demo*

# Case Studies

Applied the clone detector to Wrangler itself and other Erlang applications with the threshold value of 30 for the minimum number of tokens and 2 for the minimum number of members in a class.

	Wrangler	Mnesia	Yaws
No. of files	44	38	68
Size (K Loc)	30.9	28.2	26.9
Time (Min)	<6	<3	<3
No. Clones	53	43	66
Inter-module clones	35	5	18

# Clearly a clone

```
about(State) ->
GS = State#gui_state.gs,
WH = [{width, 600}, {height, 160}],
Win = gs:window(GS, [{title, "About Dialyzer"}, {configure, true},
                    {default, editor, {bg, yellow}} | WH]),
Frame = gs:frame(Win, [{packer_x, [{stretch, 1}, {fixed, 60}], {stretch, 1}],
                   {packer_y, [{stretch, 1}, {fixed, 30}]}
                    | WH]),
Editor = gs:editor(Frame, [{pack_x, {1,3}}, {pack_y, 1},
                          {font, {courier, 12}}, {vscroll, right},
                          {wrap, word}]),
Button = gs:button(Frame, [{label, {text, "Ok"}}, {pack_xy, {2,2}}],
gs:config(Win, {map, true}),
gs:config(Frame, WH),

AboutFile = filename:join([code:lib_dir(dialyzer), "doc", "about.txt"]),
case gs:config(Editor, {load, AboutFile}) of
{error, Reason} ->
  gs:destroy(Win),
  error(State,
    io_lib:format("Could not find doc/about.txt file!\n\n ~p",
      [Reason]));
ok ->
  gs:config(Editor, {enable, false}),
  TopWin = State#gui_state.top,
  show_info_loop(TopWin, Win, Frame, Button)
end.
```

```
help_warnings(State) ->
GS = State#gui_state.gs,
WH = [{width, 600}, {height, 500}],
Win = gs:window(GS, [{title, "Dialyzer Warnings"}, {configure, true},
                    {default, editor, {bg, white}} | WH]),
Frame = gs:frame(Win, [{packer_x, [{stretch, 1}, {fixed, 60}], {stretch, 1}],
                   {packer_y, [{stretch, 1}, {fixed, 30}]}
                    | WH]),
Editor = gs:editor(Frame, [{pack_x, {1,3}}, {pack_y, 1},
                          {font, {courier, 12}}, {vscroll, right},
                          {wrap, word}]),
Button = gs:button(Frame, [{label, {text, "Ok"}}, {pack_xy, {2,2}}],
gs:config(Win, {map, true}),
gs:config(Frame, WH),

AboutFile = filename:join([code:lib_dir(dialyzer), "doc", "warnings.txt"]),
case gs:config(Editor, {load, AboutFile}) of
{error, Reason} ->
  gs:destroy(Win),
  error(State,
    io_lib:format("Could not find doc/warnings.txt file!\n\n ~p",
      [Reason]));
ok ->
  gs:config(Editor, {enable, false}),
  TopWin = State#gui_state.top,
  show_info_loop(TopWin, Win, Frame, Button)
end.
```

From the dialyzer graphical user interface ... clearly ready for abstraction and reuse.

# Less clearly worth replacing

```
OkButton = gs:button(WinPacker, [{label, {text, "Ok"}},  
                             {pack_xy, {2,3}}]),
```

```
CancelButton = gs:button(WinPacker, [{label, {text, "Cancel"}},  
                                    {pack_xy, {3,3}}]),
```

Also from the dialyzer GUI ... would it be clearer to have an intervening common function call?

# Related Work

Existing clone detection approaches:

- Program text-based.
- Token-based.
- AST-based.
- PDG-based.
- Hybrid approaches.

Language dependent or independent?

# Future work

Use visualization techniques to improve the presentation of clone results.

Extend the current approach to find “similar” code fragments.

How to automate or semi-automate the workflow of clone detection and removal.

# Conclusions

The Wrangler clone detector

- Relatively efficient
- No false positives.

Refactorings support interactive removal of clones.

Integrated in the development environment.

*Questions?*

*[www.cs.kent.ac.uk/projects/refactor-fp/](http://www.cs.kent.ac.uk/projects/refactor-fp/)*