Practical Analyses for Refactoring Tools

Simon Thompson
University of Kent, UK

Huiqing Li
EE, UK

Context

- Building refactoring tools for functional programming languages.
- Haskell, OCaml, CakeML, …
- Wrangler, a refactoring tool for Erlang.

Wrangler

- Structural, process, macro, … refactorings.
- Automate the simple; support the complex.
- “Code smell” inspection: e.g. clone detection and elimination.
- Extensible with API/DSL

Refactoring

Refactoring

```
loop_a() ->
    receive
      stop -> ok;
      {msg, _Msg, 0} -> loop_a();
      {msg, Msg, N} ->
        io:format("ping!~n"),
        timer:sleep(500),
        b ! {msg, Msg, N - 1},
        loop_a()
     end.
```

Refactoring

```
loop_a() ->
    receive
      stop -> ok;
      {msg, _Msg, 0} -> loop_a();
      {msg, Msg, N} ->
        io:format("ping!~n"),
        timer:sleep(500),
        b ! {msg, Msg, N - 1},
        loop_a()
     end.
```

```
loop_a() ->
    receive
      stop -> ok;
      {msg, _Msg, 0} -> loop_a();
      {msg, Msg, N} ->
        body(Msg,N),
        loop_a()
     end.

body(Msg,N) ->
        io:format("ping!~n"),
        timer:sleep(500),
        b ! {msg, Msg, N - 1}.
```

Refactoring

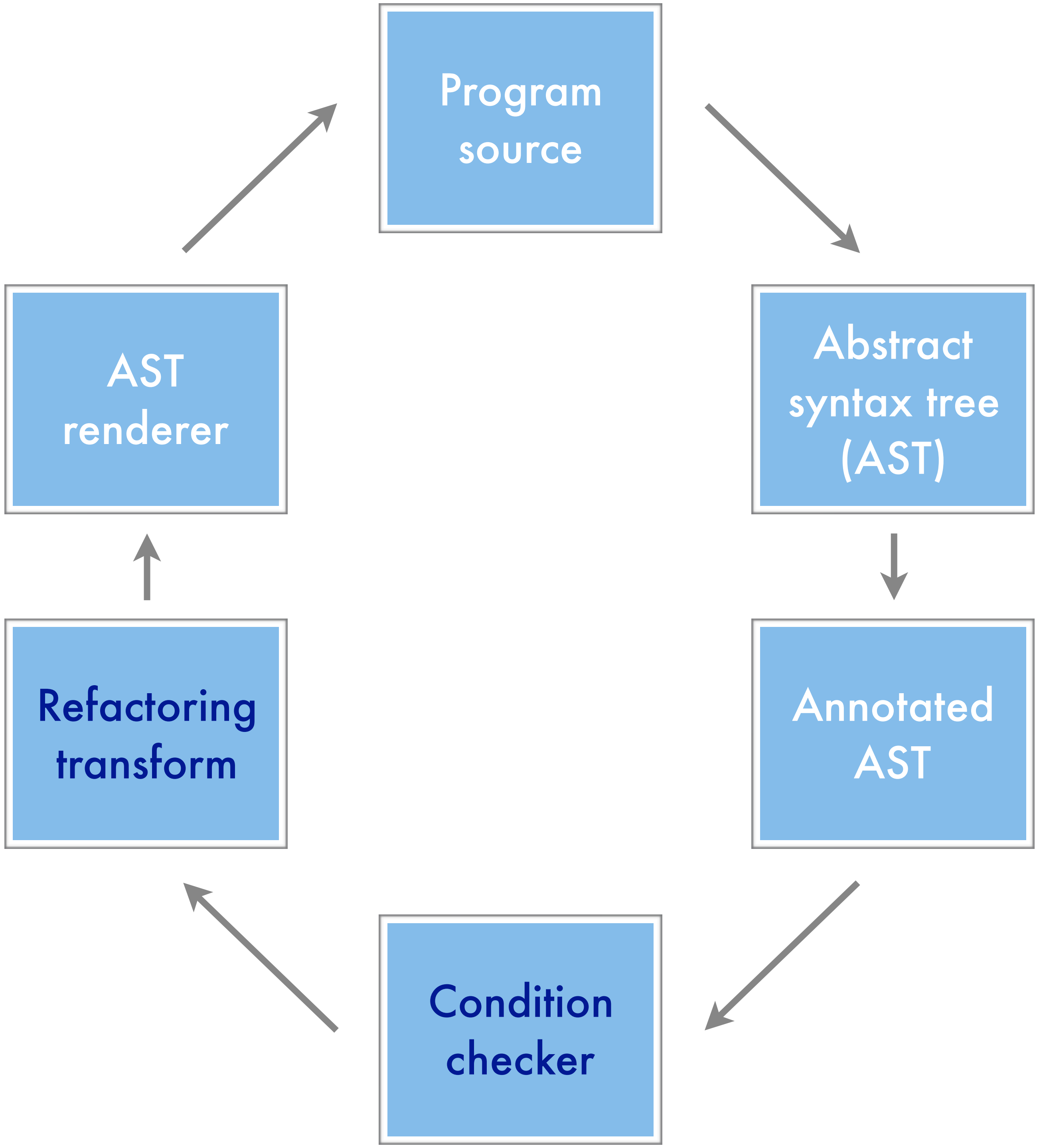
- Refactorings are diffuse and bureaucratic.
- Transformation + pre-condition
- Not just syntax: static semantics, types, modules, macros …
- Users must trust and be comfortable.

User requirements

- Target the full language … e.g. macros.
- Integrate with editors, IDEs, test tools, …
- Preserve layout and comments.
- Preview, undo, …
- Decision support: what do I do now?

Implementation

Architecture



Program source

Abstract syntax tree (AST)

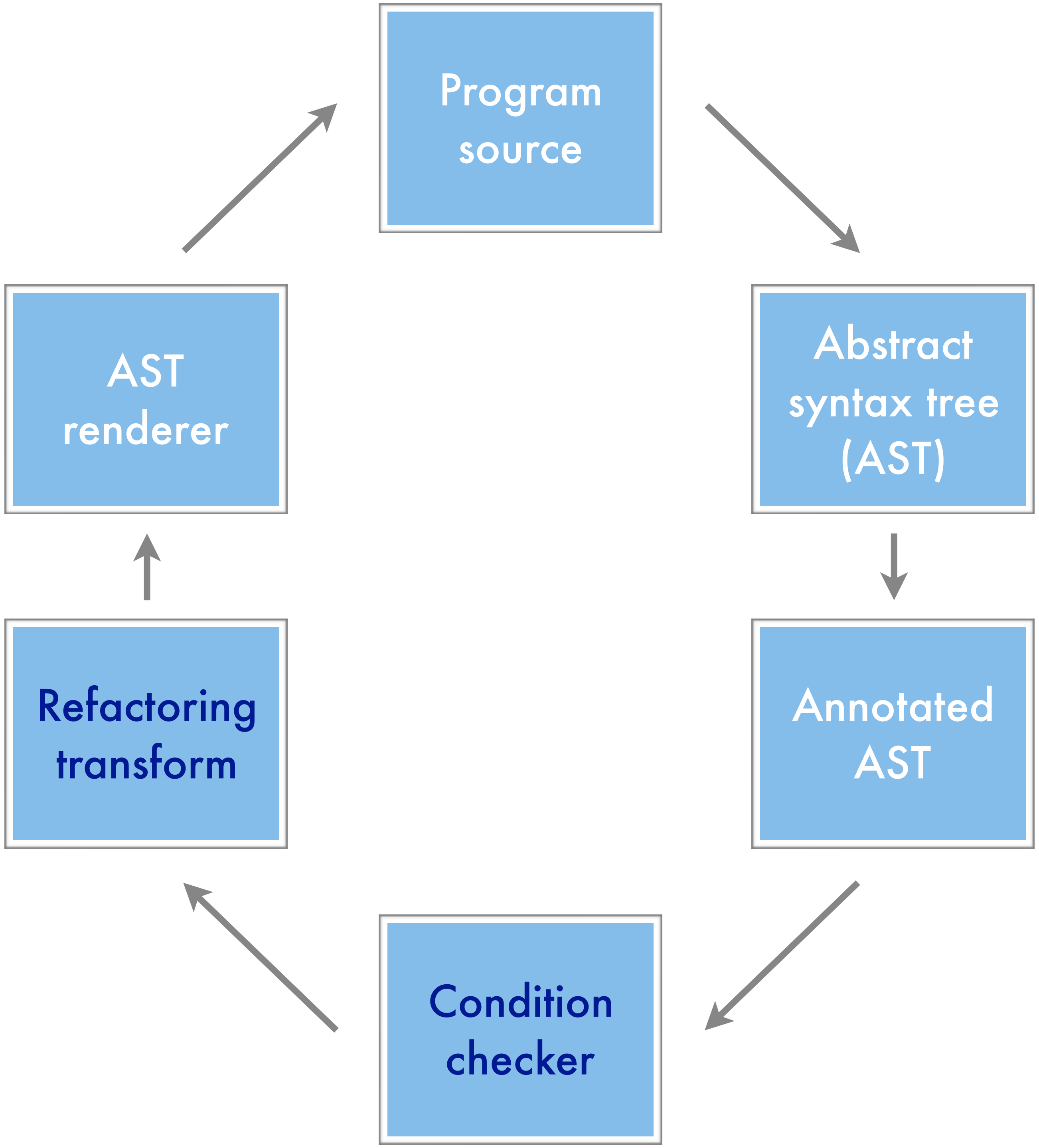
Annotated AST

Condition checker

Refactoring transform

AST renderer

Wrangler



Program source

Abstract syntax tree (AST)

Annotated AST

Condition checker

Refactoring transform

AST renderer

pretty print library

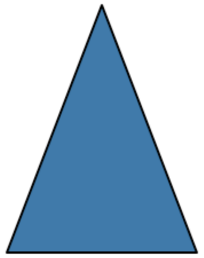
hand-written

standard Erlang parser

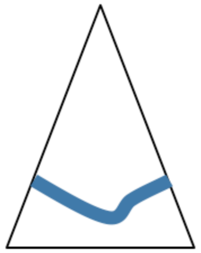
syntax_tools library ++

What is a refactoring?

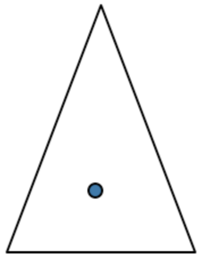
- Function on annotated ASTs, using
  - names: function, module, …
  - position of current focus,
  - current selection,
  - interactively info: Y/y/N/…



full



stop



one

Analysis

Static semantics

- Will be different in different languages.
  - Bound variables in patterns.
  - Multiple binding occurrences.
- What hope for a generic tool?

```
receiveFrom(Pid) ->
  receive
    {Pid,Payload} -> …
    … -> …
  end.
```

```
foo(Z) ->
  case Z of
    {foo,Foo} -> X=37;
    {bar,Bar} -> X=42
  end,
  X+1.
```

Types

- Monomorphic arguments and generalisation.
- Dealing with type declarations.
- Erlang: do we respect the “intended” type?

```
foo({Pid,Payload}) ->
  Payload+1.
```

```
foo(Z) ->
  Z#msg.payload+1;
```

```
foo({Pid,Payload}) ->
  Payload+1.
```

Modules

- Haskell: need call graph from import and export.

```
module Server where
import Messaging

processMsg z =
  format(msg(z))
```

- Erlang: convention is to make explicit calls to other modules.

```
-module(server).
-export([processMsg/1]).

processMsg(Z) ->
  Msg = messaging:msg(Z);
  format(Msg).
```

Side-effects

- Know the side-effects of all BIFs.
- Propagate through the call graph.
- Wrap side-effecting expressions in a `fun` when generalising.

```
printList(0) -> true;
printList(N) ->
  io:format("*"),
  printList(N-1).

printlist(3).
```

```
printList(F,0) -> true;
printList(F,N) ->
  F(),
  printList(F,N-1).

printlist(
 fun()->io:format("*") end,3).
```

Atom analysis

- Identifiers are atoms.
- The atom `foo` used as
  - Module name
  - Function name
  - Process name
  - Just an atom

```
-module(foo).

start() ->
  Pid = spawn(foo,foo,[foo]),
  register(foo,Pid) …

foo(X) -> …
```

Process structure

- Erlang processes identified by pids.
- Trace value of `Pid` through variables.
- Replace use of `Pid` by named process.

```
-module(foo).

start() ->
  Pid = spawn(foo,foo,[foo]),
  foo(Pid).

foo(Pid) ->
  … Pid …,
  bar(Pid),
  ….
```

Frameworks: OTP

- Respect the callback interface in use of OTP behaviours.

```
init(FreqList) ->
  Freqs = {FreqList, []},
  {ok, Freqs}.

terminate(_,_) ->
    ok.

handle_cast(stop, Freqs) ->
  {stop, normal, Freqs}.

handle_call(allocate, From, Freqs)
 ->
  {NewFreqs, Reply} =
      allocate(Freqs, From),
  {reply, Reply, NewFreqs};
```

Frameworks: testing

- Conventions for unit tests in EUnit.
- Use of macros in EUnit and Quviq QuickCheck.

```
-module(serial).
-include_lib("eunit/include/eunit.hrl").
-export([treeToList/1, listToTree/1,
         tree0/0, tree1/0,]).

treeToList(Tree) -> …
```

```
-module(serial_tests).
-include_lib("eunit/include/eunit.hrl").
-import(serial, [treeToList/1, listToTree/1,
                 tree0/0, tree1/0,]).

leaf_test() ->
  ?assertEqual(tree0() ,
               listToTree(treeToList(tree0()))).
```

Clone detection

- Common generalisation?
- Extract into a function.
- Choosing threshold parameters for detection.
- No “eliminate all clones” button … need domain knowledge.

```
loop_a() ->
    receive
     {msg, _Msg, 0} -> ok;
     {msg, Msg, N} ->
         io:format("ping!~n"),
         b ! {msg, Msg, N-1},
         loop_a()
        end.

new_fun(Msg,N,New_Var1,New_Var2) ->
         io:format(New_Var1),
         New_Var2 ! {msg, Msg, N-1}.

loop_b() ->
    receive
     {msg, _Msg, 0} -> ok;
     {msg, Msg, N} ->
         io:format("pong!~n"),
         a ! {msg, Msg, N-1},
         loop_b()
    end.
```

Other “bad smells”

- Modularity smells
  - Move function(s) between modules
  - Split/merge modules
- Decision support desirable

Approach

Pragmatic

- 90% is better than 0%.
- The last 10% from the user …
- … or fixed manually, using compiler.

Persistent

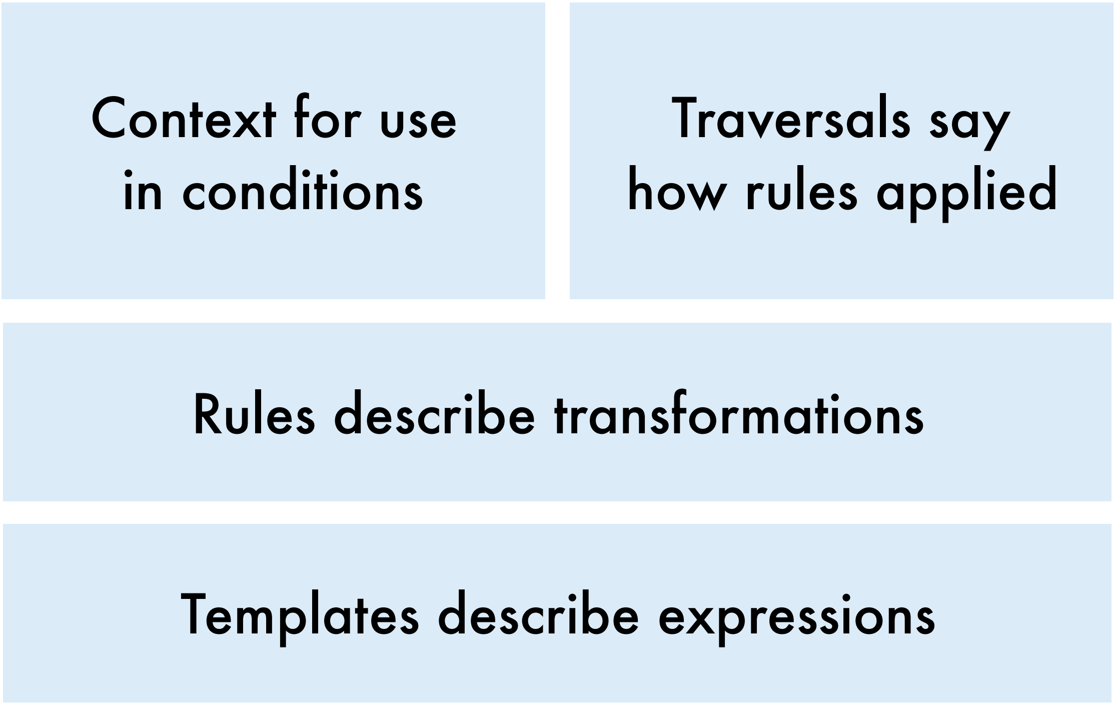
- Maintain representation alongside the text, or re-parse and analyse each time?
- Allow some structure to persist, e.g. module dependency graphs.
- Erlang concurrency makes this easy …
- … and potentially more efficient.

Incremental

- Clone detection made incremental.
- Can run with “nightly build”.
- Preserve information at function level.

Extensible

- Allow users access to the internal libraries, with a higher-level API.
- New refactorings and analyses.
- Script for composite refactorings: DSL.



Context for use in conditions

Traversals say how rules applied

Rules describe transformations

Templates describe expressions

Approach

- Pragmatic
- Persistent
- Incremental
- Extensible
- Single language

Drawbacks

- Single language?
- *Ad hoc*
- Refactoring representation
- Textual representation

Thanks

- EPSRC; EU FP 7
- Claus Reinke
- Chris Brown
- Nik Sultana
- Richard Carlsson …
- … + other githubbers

- Gyorgy Orosz
- Melinda Toth
- Dániel Horpácsi
- Dániel Drienyovszky
- Adam Lindberg
- Judit Kőszegi

Questions?