Building Refactoring Tools for Functional Languages

Simon Thompson and Huiqing Li
University of Kent, UK

Overview

- Erlang for Haskellers
- Refactoring
- The tools
- Design, analysis and implementation
- Extensions
- Reflections and future plans

Erlang for Haskellers

Weakly typed

- Numbers, atoms, tuples and lists.
- (Extensible) records: syntactic sugar.
- Dynamic aspects.

```
Val = [12,”34”,[56],{[78]}].
```

```
NewTree =
  Tree#tree{value=42}.
```

```
F = list_to_atom(“blah”),
apply(?MODULE,F,Args).
```

Concurrency at the core

- Processes.
- No shared memory.
- Asynchronous message passing.
- Process ids or names.

```
Pid = spawn(server,fac,[]),
Pid ! {self(),N},
receive
  {ok,Result} -> …
  stopped     -> …
end, …
```

```
fac() ->
  receive
    {From, stop} ->
      From ! stopped;
    {From, N} ->
      From ! {ok,fact(N)},
      fac()
  end.
```

Pattern Matching

- Haskell-style, but …
- Single assignment.
- Bound variables can appear in patterns.
- Selective receive.

```
N = 46,
N = 23+23,
N = 35,
 …
```

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

```
receive {foo,Foo} -> … end,
receive {bar,Bar} -> … end …
```

Open Telecom Platform

- Erlang + OTP.
- Design patterns.
- Generic behaviours.
- Server, FSM, event handler, supervisor.
- Callback interface.

```
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};
```

Other Erlang features

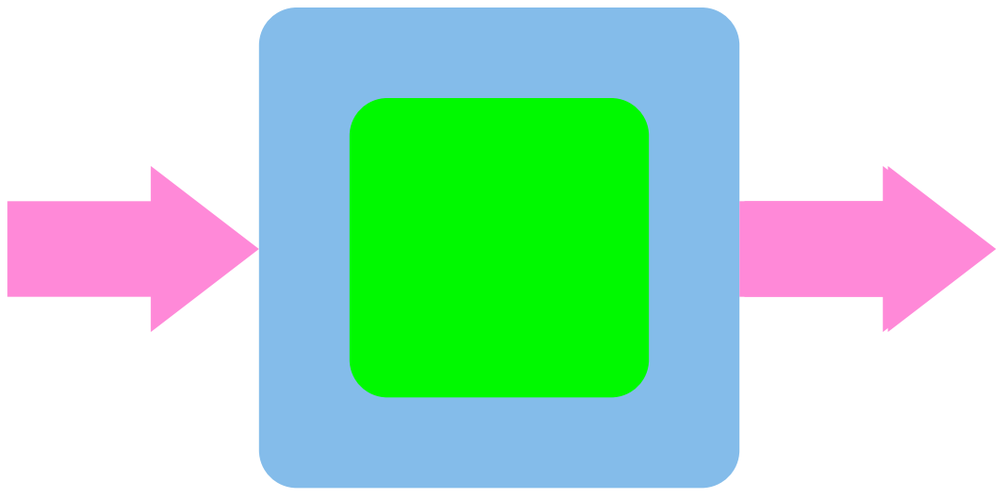
- Eager evaluation.
- Side effects.
- Name/arity identify a function.
- Bindings: shadows, multiple BOs.
- Macros.
- Conventions: OTP, EUnit, QuickCheck

Pragmatics

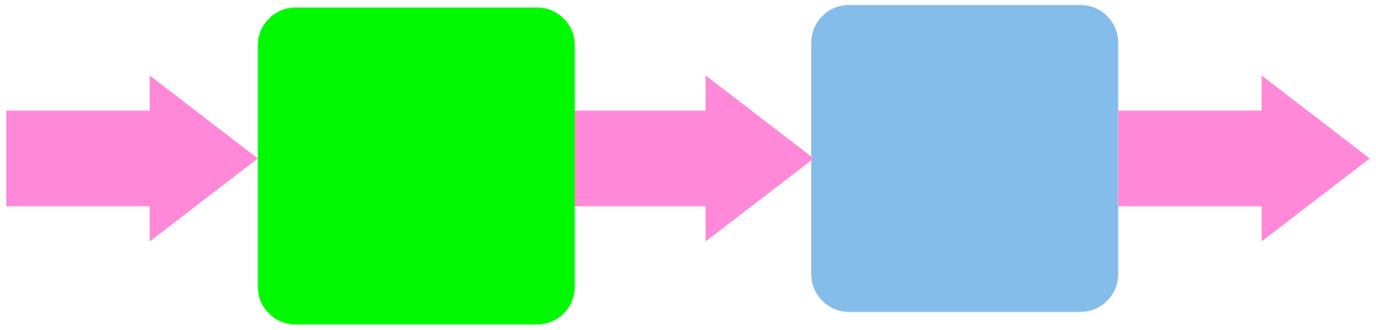
- One implementation, one standard.
- Well-defined, controlled release cycle.
- Open Source but … Ericsson effort.
- Erlang Extension Proposals.

Refactoring

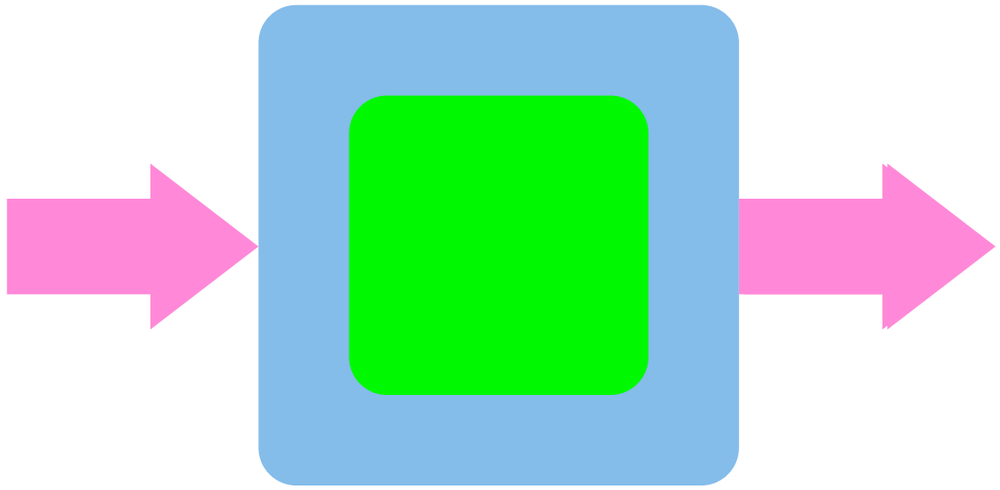
Refactoring



Refactoring



Refactoring



Erlang example

```
-module (test).
-export([f/1,add_one/1]).

add_one([H|T]) ->
  [H+1 | add_one(T)];

add_one([]) -> [].

f(X) -> add_one(X).
```

Generalisation

```
-module (test).
-export([f/1,add_one/2]).

add_one([H|T],N) ->
  [H+N | add_one(T,N)];

add_one([],_) -> [].

f(X) -> add_one(X,1).
```

Renaming

```
-module (test).
-export([f/1,add_int/2]).

add_int([H|T],N) ->
  [H+N | add_int(T,N)];

add_int([],_) -> [].

f(X) -> add_int(X,1).
```

Haskell example

```
data Tr a
  = Leaf a |
    Node (Tr a) (Tr a)
```

```
flatten :: Tr a -> [a]

flatten (Leaf x) = [x]
flatten (Node s t) = flatten s
                     ++ flatten t
```

data to abstract type

```
data Tr a
  = Leaf {leaf::a} |
    Node {left,right::Tr a}

isLeaf = …       mkLeaf = …
isNode = …       mkNode = …

flatten :: Tr a -> [a]
flatten t
 | isleaf t = [leaf t]
 | isNode t = flatten (left t)
              ++ flatten (right t)
```

Refactoring ≠ Transformation

- Traditional program transformations often work over a single definition.
- Refactorings diffuse and bureaucratic … … so tedious and error-prone by hand.
- Not just editing: static semantics, types, modules, macros … layout, comments.
- Results must be read by programmers.

Systems

HaRe

- Full Haskell 98 coverage.
- Structural and data refactorings.
- Clone detection and elimination.
- Programmatica and Strafunski used.
- Integrated within Vim and Emacs.

Wrangler

- Structural, process, macro refactorings.
- “Code smell” inspection.
- Similar code detection and elimination.
- Test-awareness; testing refactorings.
- Integrated within Emacs and Eclipse.

Wrangler demo

Aquamacs File Edit Options Tools Refactor Inspector QuickCheck Erlang Window Help

New Open Recent Revert Save Help

scratch 1 brchcp_vig_calls_SUITE.erl 2

```
%% Code testing frequency.erl which is itself fro
%%   Erlang Programming
%%   Francecso Cesarini and Simon Thompson
%%   O'Reilly, 2008
%%   http://oreilly.com/catalog/9780596518189/
%%   http://www.erlangprogramming.org/
%%   (c) Francesco Cesarini and Simon Thompson

-module(frequency_tests).
-include_lib("eunit/include/eunit.hrl").
-import(frequency,[start/0, stop/0, allocate/0, de

%% start() and stop()

start_test_() ->
    {setup,
     fun ()  -> ok end,              % null startup
     fun (_) -> stop() end,          % stop the sy
     ?_assertMatch(true,start())     % make sure th
    }.

stopFirst_test_() ->
    {setup,
     fun ()  -> ok end,              % null startup
     fun (_) -> ok end,              % no cleanup
     ?_assertError(badarg,stop())    % stop before
    }.

startStop_test_() ->
    {setup,
     fun ()  -> start() end,         % start normally!
```

-:-- frequency_tests.erl Top (7,46) (Erlang)

Rename Variable Name
Rename Function Name
Rename Module Name
Generalise Function Definition
Move Function to Another Module
Function Extraction
Fold Expression Against Function
Tuple Function Arguments
Unfold Function Application

Introduce a Macro
Fold Against Macro Definition

Detect Identical Code in Current Buffer
Detect Identical Code in Dirs
Identical Expression Search
Detect Similar Code in Current Buffer
Detect Similar Code in Dirs
Similar Expression Search

Refactorings for QuickCheck ▶
- Introduce ?LET
- Merge ?LETs
- Merge ?FORALLs
- eqc_statem State to Record
- eqc_fsm State to Record
- gen_fsm State to Record

Process Refactorings (Beta) ▶
Normalise Record Expression

Undo C-c C-_

Customize Wrangler

Version

Eclipse File Edit Refactor Navigate Search Project Run Window Help

Rename module... ⇧⌥R M
Rename process...
Rename function... ⇧⌥R F
Rename variable... ⇧⌥R V

Detect duplicated code...
Search expression...

Convert Function to process...
Move function... ⇧⌥V F
Fold expression...
Extract function...
Generalise function...

Tuple function parameters...

e SDK – /Users/simonthompson/Documents/workspace

Resource

Erla

External_File
test

syntax.

```
% Form -> Form

makeNeg(N) -> #neg{neg = N}.

% String -> Form

makeLeaf(L) -> #leaf{leaf = L}.

% Derived constructors for => and <=>

% (Form,Form) -> Form

makeImp(L, R) -> makeDisj(makeNeg(L), R).

% (Form,Form) -> Form

makeIff(L, R) -> makeConj(makeImp(L, R), makeImp(R, L)).

% Print a formula to the output.

% Form -> ()

printFormula({conj, L, R}) ->
    io:format("("),
    printFormula(L),
    io:format("/\\"),
    printFormula(R),
    io:format(")");
```

Outline

- module: syntax
- export
- record_definition: conj
- record_definition: disj
- record_definition: leaf
- record_definition: neg
- form1/0
- form2/0
- makeConj/2 (L, R)
- makeDisj/2 (L, R)
- makeIff/2 (L, R)
- makeImp/2 (L, R)
- makeLeaf/1 (L)
- makeNeg/1 (N)
- printFormula/1
  - ({conj, L, R})
  - ({disj, L, R})
  - ({neg, N})
  - ({leaf, L})
- showFormula/1
  - ({conj, L, R})
  - ({disj, L, R})
  - ({neg, N})
  - ({leaf, L})
- simplify/1
- test1/0
- test2/0

Writable | Smart Insert | 26 : 2

Top-level design

Comprehensive

- Target the full language.
  - Haskell 98.
  - Erlang/OTP R12, R13.

Accessible to the user

- Integrate with the principal IDEs …
  - Vim, Emacs and Eclipse.
- … and other parts of the tool chain.
  - Test frameworks, documentation systems, build infrastructure, … ?

Readable

- Preserve layout.
  - Automated layout.
    - Layout style inference.
- Preserve comments.
  - Conventions / heuristics.

Extensible

- API for user-defined refactorings.
  - In the host language.
- A DSL for refactoring?

What every user wants

- Preview.
- Undo.
- My favourite refactoring, please.
- Assistance in finding and applying refactorings.

Design experience

What do you mean?

- Generalise on 1.

```
-module (setup).
-export([port/1]).

port() ->
  PortId    = 1,
  SessionId = 127+1,
  Version   = 1,
  {PortId,SessionId,Version}.
```

- One, some or all occurrences of 1?
- One or all clauses?

Compensate or reject?

- Lift g to a top-level definition.

```
f x = x + g x
      where g x = x + con
                  con = 37
```

- Fail because con not defined at top level?
- Add a parameter to g, passing in con?
- Lift con to the top-level too?

Backwards compatibility?

- Generalise over 1.
- Include a legacy version of add_one?
- Let it fail when it's called?

```
-module (test).
-export([add_one/1]).

add_one([H|T]) ->
  [H+1 | add_one(T)];
add_one([]) -> [].
```

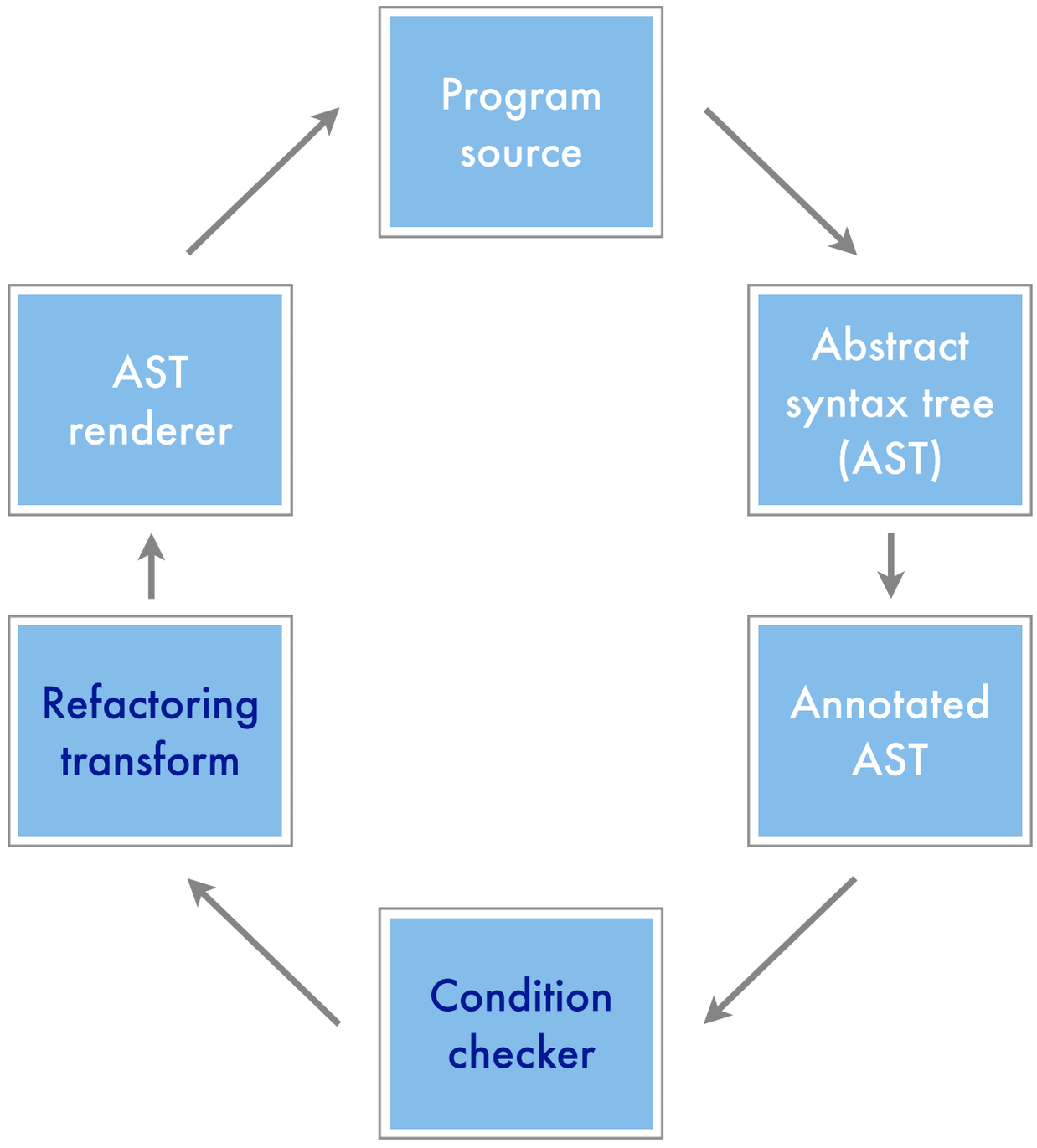
```
-module (test).
-export([add_one/1,add_one/2]).

add_one([H|T],N) ->
  [H+N | add_one(T,N)];
add_one([],N) -> [].

add_one(L) -> add_one(L,1).
```

Implementation

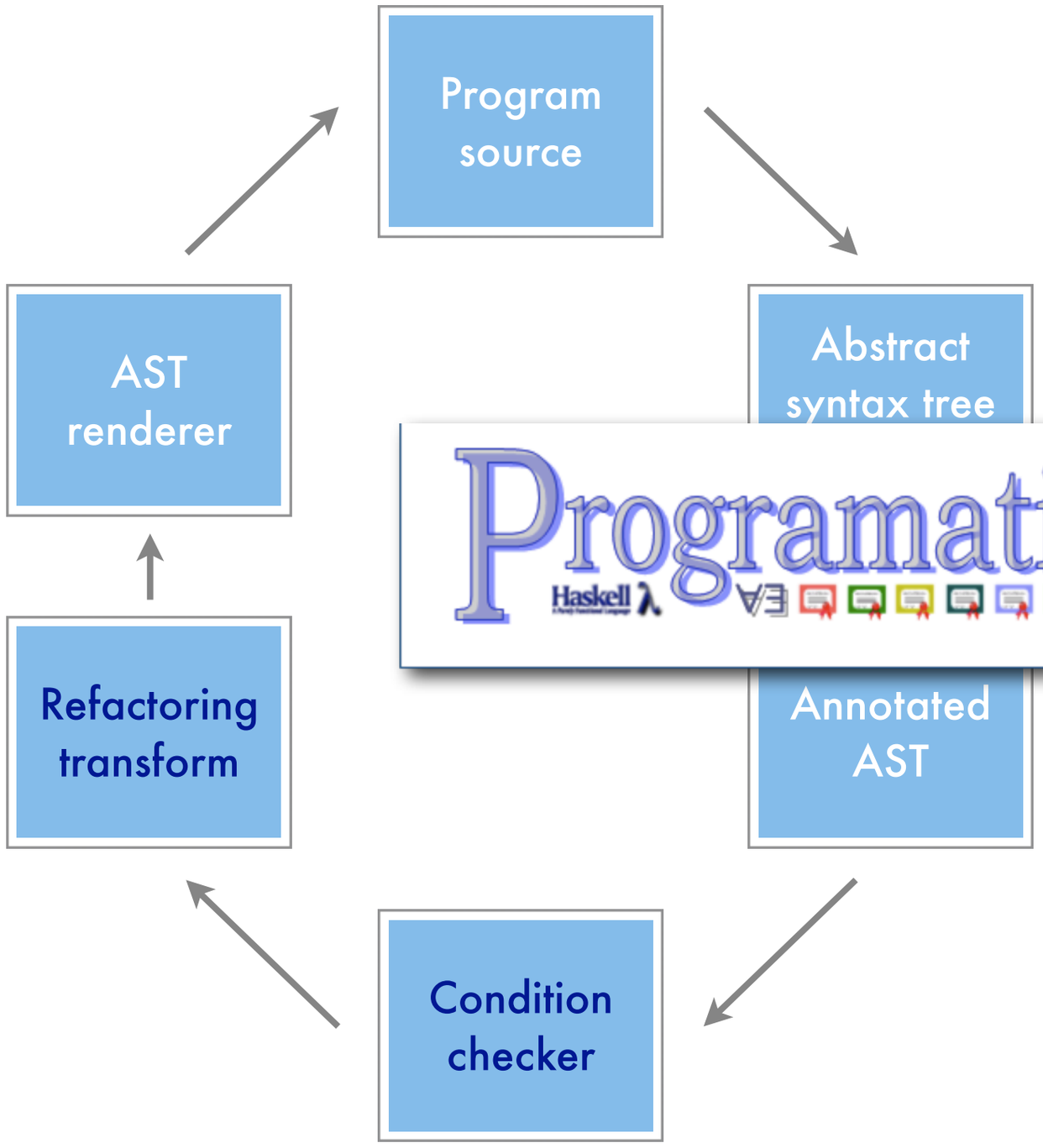
Architecture



HaRe

hand-written

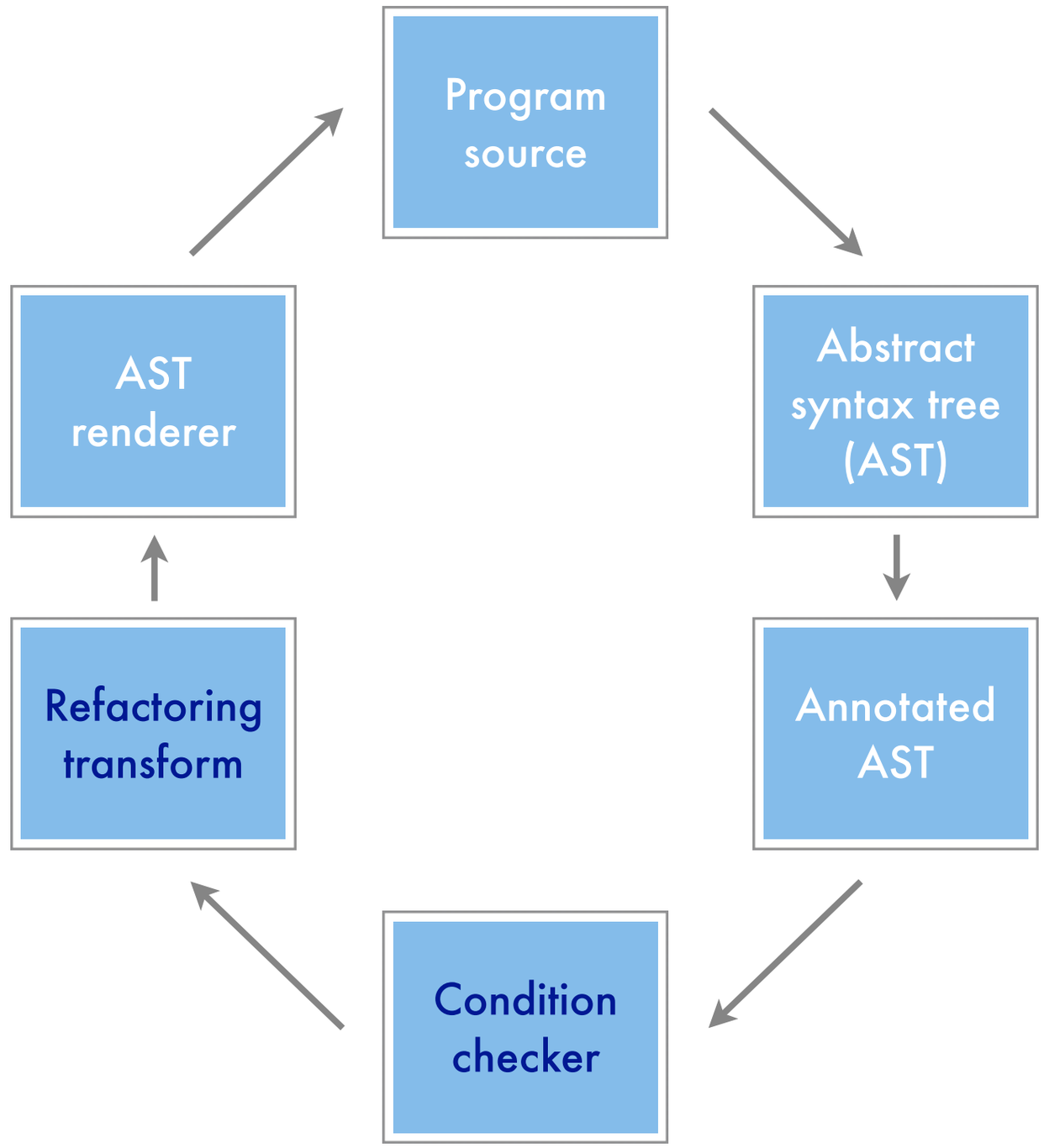
Strafunski



Programatica

Integrating Programming, Properties, and Validation

Wrangler



pretty print library

standard Erlang parser

hand-written

syntax_tools library ++

Do it yourself?

- Use other frameworks if possible …
  but you may have to maintain them.
- DIY? Get complete control, but can certainly be maintenance problems.
- Existence and stability of the right APIs within compilers?

Representation

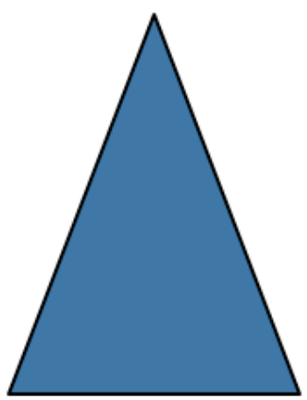
- A refactoring is a Haskell/Erlang function on AASTs, parameterised by
  - names: function, module, …
  - position of current focus,
  - current selection,
  - interactively gathered Y/y/N/…

Alternative representations

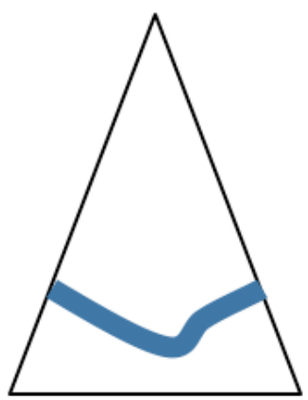
- Better representation of position?
  - Name/logical position in tree.
  - Easier scripting of sequences.
- Generate a set of diffs, in some form?
  - More direct interface with Eclipse.
  - Fits with darcs? Commutativity?

Traversals

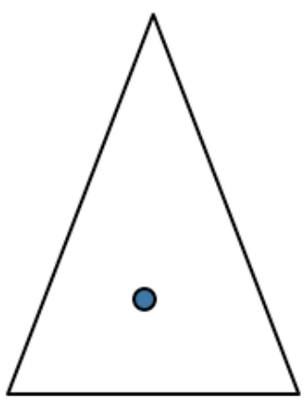
- Condition checks and transformations use multi-sorted tree traversals.
- Haskell: use one of the generics libraries.
- Erlang: write it yourself.



`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

- Erlang 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.
- Use case: replace use of `Pid` by a 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()))).
```

Persistence?

- Maintain representation alongside the text, or re-parse and analyse each time?
- Speed / complication tradeoff.
- Allow some structure to persist, e.g. module dependency graphs.
- Erlang processes readily support internal persistence.

Integration

Emacs

- LISP inside: ease of programming.
- Erlang and Haskell modes.
- Portable across platforms.
- No intrinsic notion of project.
  - Problems with multi-module undo.
- Emacs vs XEmacs.

Eclipse

- Java inside: ease of programming?
- ErlIDE plugin: Wrangler integrated.
- Portable across platforms.
- Integrated: project, build, test etc.
- Eclipse refactoring API limited.
- Different audience to that of Emacs.

Vim

- Difficult to program.
- Not portable across platforms: e.g. different models for external processes.
- Projects: similar problems to Emacs.
- We didn't try to integrate Wrangler …

Extensions

Clone detection

- Common generalisation?
- Extract into a function.
- Choosing threshold parameters for detection.
- No “eliminate all clones” button … need domain knowledge.
- PEPM’09, ’10, PADL’10.

```
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'

- Local properties
  - Depth of nesting of receive or case.
  - Size of functions or modules.
- Modularity smells
  - Move function(s) between modules
  - Split/merge modules

How to test?

- Build unit test suite by hand …
- … or use random testing?
  - Generate random programs using a simple attribute grammar.
  - Refactor with a random refactoring
  - Generate program inputs randomly.
  - Test old(inputs) $\stackrel{?}{=}$ new(inputs).

Reflections

Language flaws

Haskell

- No hiding on export.
- Field names for standard types?
- Tab is a real nightmare.

Erlang

- No types.
- No processes or channel explicitly.
- Inconsistency in binding patterns.
- Multiple roles of atoms

What to support?

- Automate basic refactorings.
- Semi-automation for more complex reports and refactorings.
- Many more specialised refactorings will never be implemented.
- “RISC vs CISC”: do simple things well.

Past and present obstacles

- We don't support GHC Haskell.
- We don't support editor X.
- Over-complicated installation and dependencies.
- Lack of support for "smell detection".
- General question of trust?

Future plans

- Revisit the refactoring DSL question.
- More tools to support and guide the user.
- Refactoring and testing
  - Property discovery from tests, clones.
  - Refactoring tests themselves.
- Revisit a refactoring tool for GHC?

Thanks

- Claus Reinke
- Chris Brown
- Nik Sultana
- Gyorgy Orosz
- Melinda Toth
- Dániel Horpácsi
- Dániel Drienyovszky
- Adam Lindberg

- EPSRC
- EU FP 7

Questions?