Point Of No Local Return:

The Continuing Story Of Erlang Type Systems

Zeeshan Lakhani
Papers We Love, Basho Technologies
@zeeshanlakhani

I don't know nothin

I don't know nothin

- *Konstantinos Sagonas*

I don't know nothin

- *Konstantinos Sagonas*
- *John Hughes*

I don't know nothin

- *Konstantinos Sagonas*
- *John Hughes*
- *Joe Armstrong*

I don't know nothin

- *Konstantinos Sagonas*
- *John Hughes*
- *Joe Armstrong*
- *Tobias Lindahl*

I don't know nothin

- *Konstantinos Sagonas*
- *John Hughes*
- *Joe Armstrong*
- *Tobias Lindahl*
- *Maria Christakis*

I don't know nothin

- *Konstantinos Sagonas*
- *John Hughes*
- *Joe Armstrong*
- *Tobias Lindahl*
- *Maria Christakis*
- *Joe Devivo*

I don't know nothin

- *Konstantinos Sagonas*
- *John Hughes*
- *Joe Armstrong*
- *Tobias Lindahl*
- *Maria Christakis*
- *Joe Devivo*
- *more...*

ATLANTIC

45 R.P.M.

45-2819

Pub., Cotillion,
BMI
Time: 3:30

VOCAL
A-2242S SP

YOUR MOVE

(Anderson)

YES

Produced by Yes & Eddie Offord
From Atlantic LP 8283

MFG. BY ATLANTIC RECORDING CORP., 1841 BROADWAY, NEW YORK, N.Y.

Don't surround yourself with yourself,
Move on back two squares,
Send an Instant Karma to me,
Initial it with loving care
Don't surround
Yourself.
'Cause it's time, it's time in time with your time and its news is captured
For the queen to use.

Static Strong Type System - SML

```
datatype suit = Clubs | Diamonds | Hearts | Spades
datatype rank = Jack | Queen | King | Ace | Num of int
type card = suit * rank

fun card_color card =
    case card of
        (Clubs, _) => Black
      | (Spades, _) => Black
      | (Diamonds, _) => Red
      | (Hearts, _) => Red
```

Dynamic Strong Typing[30]

```
> 6 + "1".
** exception error: an error occurred when
evaluating an arithmetic expression
     in operator  +/2
called as 6 + "1"
```

01110101
10011101
11110111
00010011

Jaden Smith Coding @jaden_coding · 23 Feb 2015

I Hope **Erlang** Wins An Oscar. Best **Type System**.

In reply to Tim Dysinger

Martin Kristiansen @fold_right · 5 Oct 2014

@dysinger dear **Erlang** can I plz has a strong **type system** ;)

View conversation

Cons T Åhs @lisztspace · Feb 15

@jlouis666 agreed, as long as it is reasonable. Bolting on a type system afterwards is always quite challenging.

In reply to LuaJIT To Quit

VP, Unpopular Things @potsdamnhacker · 11 Jun 2015

@warrenhenning @olix0r I would argue **Erlang** + **Dialyzer** is superior to Go's **type system**, but optional.

View conversation

Jesper L. Andersen @jlouis666 · Jan 15

To give **Erlang** a **type system** requires you to solve a nontrivial problem of dynamic multi-party session types first.

9 14

"Dynamic typing is but a special case of static typing, one that limits, rather than liberates, one that shuts down opportunities, rather than opening up new vistas. Need I say it?"[24]

— *Bob Harper*

"All is fair in love and war, even trying to add a static type system in a dynamically typed programming language"[23]

— *Lindahl and Sagonas*

Gradual Typing

"Siek and Taha [2006] coined the term gradual typing to describe a theory for integrating static and dynamic typing within a single language that 1) puts the programmer in control of which regions of code are statically or dynamically typed and 2) enables the gradual evolution of code between the two typing disciplines."[29]

— *Siek, et al.*

```
(struct pt ([x : Real] [y : Real]))

(: distance (-> pt pt Real))

(define (distance p1 p2)
  (sqrt (+ (sqr (- (pt-x p2) (pt-x p1)))
           (sqr (- (pt-y p2) (pt-y p1))))))
```

```
(struct pt ([x : Real] [y : Real]))

(: distance (-> pt pt Real))

(distance "foo" 4)
```

stdin::189: Type Checker: type mismatch
  expected: pt
  given: String
  in: "foo"
  context...:
   f269
    /Applications/Racket/share/pkgs/typed-racket-lib/typed-racket/typecheck/tc-app/tc-app-main.rkt:91:12: for-loop
   parse-loop559
    /Applications/Racket/share/pkgs/typed-racket-lib/typed-racket/typecheck/tc-app/tc-app-main.rkt:68:0: tc/app-regular
    /Applications/Racket/share/pkgs/typed-racket-lib/typed-racket/typecheck/tc-expr-unit.rkt:287:0: tc-expr
    /Applications/Racket/share/pkgs/typed-racket-lib/typed-racket/typecheck/tc-toplevel.rkt:560:0: tc-toplevel-form
   temp19
   /Applications/Racket/collects/racket/private/misc.rkt:87:7

I: A Subset of the Past

- $e : \tau$
- e is well-typed, meaning that its components fit together properly according to the rules (e.g., operators are applied to the right kinds of arguments), and
- τ : when e is evaluated, and its evaluation terminates, it produces a value described by τ.

Soft Typing

- Type inference applied to dynamically typed languages

- **Foundational Works**: Cartwright and Fagan's *Soft Typing*[31] & Aiken and Wimmers's *Type Inclusion on Constraints and Type Inference*[2]

- *top* type can be used in the absence of meaningful *ordinary* types;

Principal Types

- Finding a way to represent all all possible typings for a term

- **Foundational Work**: Jim's *What are principal typings and what are they good for?*[3]

- Not only a principal type but also the associated environment

- type signature only holds if the arguments in an application are subtypes of the arguments in the signature.

April 24, 1996
ISSN 1100–1836

A Prototype of a Soft Type System for Erlang

Anders Lindgren

First Runs

```
foo(X) -> [X | X].

bar([X | X]) -> X.
```

$$\texttt{foo}(\alpha) \Longrightarrow cons(\alpha, \alpha)$$
$$\texttt{bar}(cons(\alpha, \alpha)) \Longrightarrow \alpha$$

First Runs cont.

- 1996 Soft-Type system prototype by Lindgren

- Data Type (Collection) representing a mapping from variables to types, defined by Meet (GLB - combing variables in diff. expressions) and Join operations (LUB - for when inferring types with sub-clauses, like *case*)

- Constraint solver (Illyria) could not represent types dealing with individual atoms. Had issues simplifying non-canonical representations: $int \cup (int \cup float)$ to $int \cup float$

- 1998 - Armstrong/Arts - declaration files generate html pages... *the specification web*

A Practical Subtyping System For Erlang

Simon Marlow
simonm@dcs.gla.ac.uk
University of Glasgow

Philip Wadler
wadler@research.bell-labs.com
Bell Labs, Lucent Technologies

Abstract

We present a type system for the programming language Erlang. The type system supports subtyping and declaration-free recursive types, using subtyping constraints. Our system is similar to one explored by Aiken and Wimmers, though it sacrifices expressive power in favour of simplicity. We cover our techniques for type inference, type simplification, and checking when an inferred type conforms to a user-supplied type signature, and report on early experience with our prototype.

based on row variables, as introduced by Wand [Wan87], and used as the basis of the soft type system for Scheme by Cartwright, Fagan, and Wright [CF91, WC94]. It turns out that the row variable system rejects some programs that seem quite natural to us, and the circumlocutions we had to go through to construct an equivalent program that was well typed struck us as hard to explain. This isn't a problem for soft typing systems, where the goal is to improve performance by removing run-time type checking, and therefore maximum information is of greater benefit than a natural notion of typing.

The Marlow / Wadler Joint

- Wadler had a 1-year sabbatical and was going to write a type system for Erlang[11]

- Based on Aiken/Wimmers Type Inclusion Constraints and Type Inference

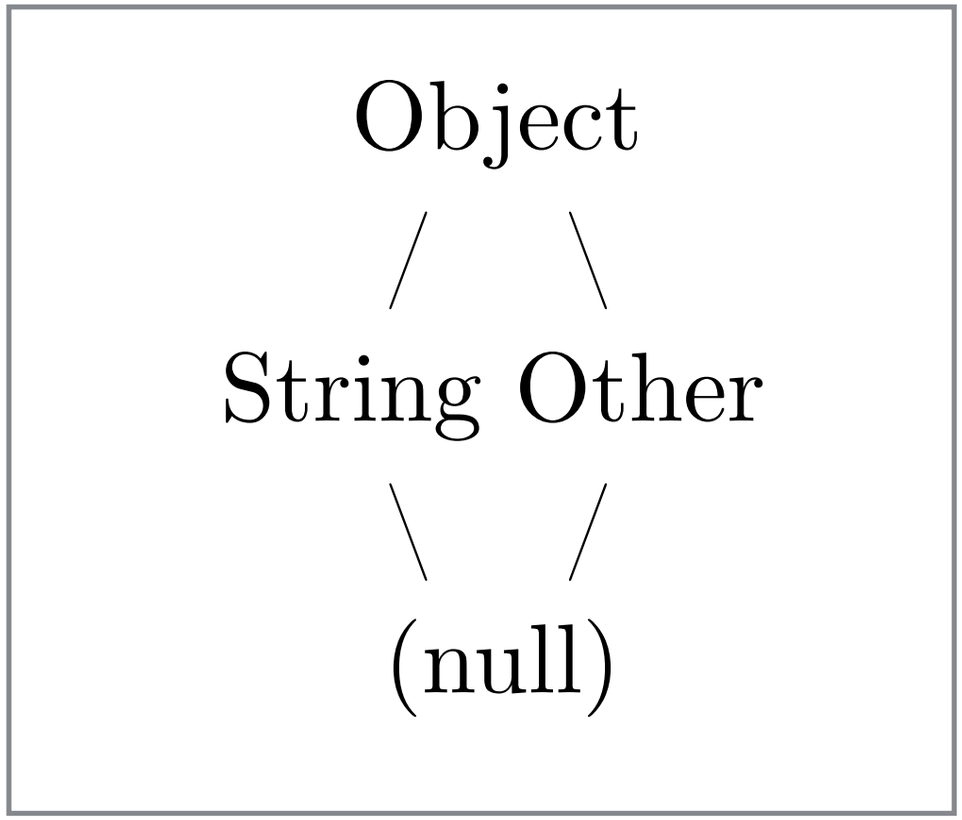
- support of recursive types and **disjoint unions**

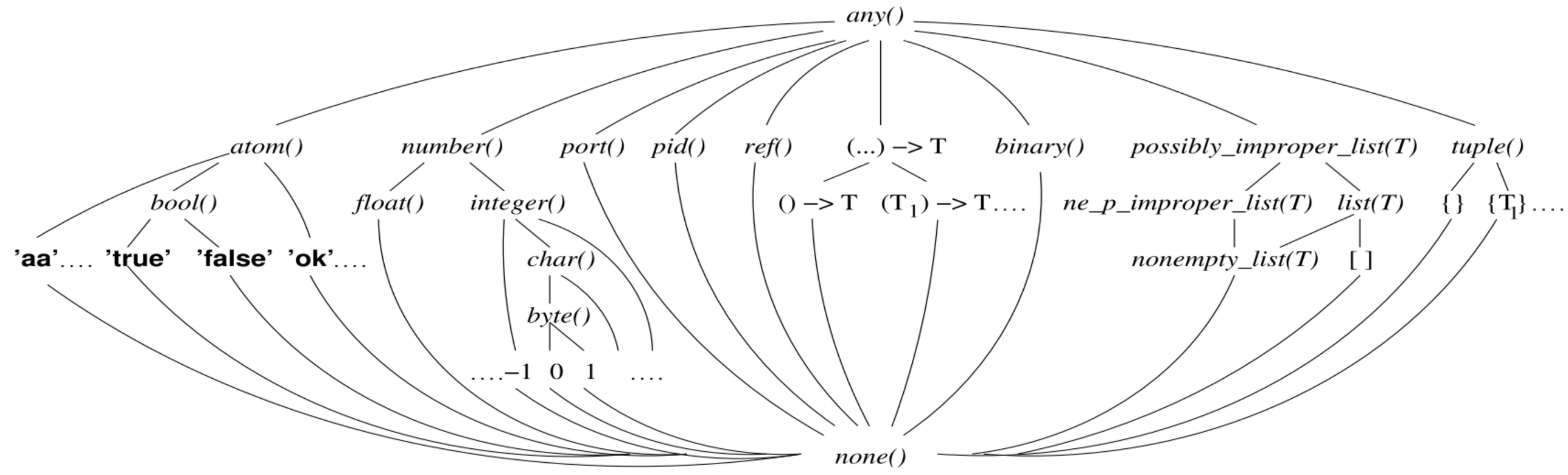
- Had type annotation system akin to Dialyzer/Typer specs

- Disappointing results: Lack of process types/inter-process checks; worked only on a subset of Erlang[11]

subtyping: try to solve sets of constraints of the form $\alpha \subseteq \beta_{[10]}$

unification (Hindley-Milner): solve constraints of the form $\alpha = \beta_{[10]}$


Object
String
Other
(null)


any()
atom()
number()
port()
pid()
ref()
(...) -> T
binary()
possibly_improper_list(T)
tuple()
bool()
float()
integer()
() -> T
(T1) -> T....
ne_p_improper_list(T)
list(T)
{}
{T1}....
'aa'....
'true'
'false'
'ok'....
char()
nonempty_list(T)
[ ]
byte()
....-1
0
1
....
none()

Unification is literally the process of looking at each of the constraints and trying to find a single type which satisfies them all[22]

To unify two type expressions is to find substitutions for all type variables that make the expressions identical

Wright/Cartwright modified Hindley-Milner typing to accommodate union types and subtyping when creating a soft typing system in Scheme[6]

Success?

```
and(true, true) -> true;
and(false, _) -> false;
and(_, false) -> false.

and(true, true) -> true;
and(false, X) -> false;
and(X, false) -> false.
```

```
and(any(), false) -> true + false.
```

```
and(X,Y) ->
    let Z = (case Y of false -> false end) in
    case X of
        true ->
            case Y of
                true -> true;
                X -> Z
            end;
        false -> false;
        X -> Z
end.
```

-spec and(_,_) -> boolean().

A soft-typing system for Erlang

Sven-Olof Nyström
Department of Information Technology,
Uppsala University, Sweden
svenolof@csd.uu.se

ABSTRACT

This paper presents a soft-typing system for the programming language Erlang. The system is based on two concepts; a (forward) data flow analysis that determines upper approximations of the possible values of expressions and other constructs, and a specification language that allows the programmer to specify the interface of a module. We examine the programming language Erlang and point to various aspects of the language that make it hard to type. We present experimental result of applying the soft-typing system to some previously written programs.

A static type system added to an existing dynamically typed programming language is sometimes referred to as a *soft-typing system*. Generally speaking, a soft typing system might serve two purposes; it can produce type information to help compiler optimizations, and it can be used, just like a static type system, to help the programmer find bugs and inconsistencies in the program.

When used as a development tool, a soft typing system can either be used when programs are written from scratch, or applied to existing programs. Any soft typing system will be sensitive to the choice of data representation and control

Another Soft Typing System

- Uses dataflow analysis to compute for each variable and subexpression in the program, an approximation of the set of possible values.

- **Generates** type expressions and **Matches** terms against expressions

- *Call(f, l, c)= c'* to allow for typed polymorphism

- Abstract, Public, Unsafe Types (mbox -> mailbox receives)

- "It turns out that specifying the interaction of an Erlang process is rather difficult"

- Similar specification language, based on Marlow/Wadler's paper, *separates out spec files from .erl files*

- *Tons of Noise (must annotate at all interface points)*

Typing Erlang

John Hughes, David Sands, Karol Ostrovský

December 12, 2002

Abstract

Type systems for concurrency express vital properties of concurrent and distributed programs, such as deadlock-freeness. These systems have grown more sophisticated with time, but unfortunately most of them apply to toy programming languages. At the other end of the spectrum, there is Erlang – an open souce functional and concurrent language with a number of successful commercial applications, but lacking a good type system. Attempts have been made to define a type system for Erlang, but these have ignored its concurrency, distribution and fault-tolerance features.

The goal of this project is to develop a type system for Erlang that can be used to successfully type check existing large Erlang programs, and used as a substrate for efficient type-based program analyses. Our approach is also inspired by recent developments of the Haskell type system, which now supports a number of advanced features (e.g. polymorphic recursion and existential types). Each of these features poses problems for type *inference*, but succumbs easily to type *checking*. Haskell's design has shown that it is possible to combine explicit type annotations, for the checkable parts, while still using type inference for most of the program. By using the same approach for Erlang we hope to combine efficiency and power.

how do we ensure that the receive expressions in a process body expect messages of the correct type?

$$\Gamma \vdash e : \tau \text{ receiving } \mu$$

makeref and guaranteeing that replies are sent to the correct process.

II: The Tao of Now

Detecting Software Defects in Telecom Applications Through Lightweight Static Analysis: A War Story

Tobias Lindahl and Konstantinos Sagonas

Computing Science, Dept. of Information Technology, Uppsala University, Sweden
{Tobias.Lindahl,Konstantinos.Sagonas}@it.uu.se

Abstract. In safety-critical and high-reliability systems, software development and maintenance are costly endeavors. The cost can be reduced if software errors can be identified through automatic tools such as program analyzers and compile-time software checkers. To this effect, this paper describes the architecture and implementation of a software tool that uses lightweight static analysis to detect discrepancies (i.e., software defects such as exception-raising code or hidden failures) in large commercial telecom applications written in Erlang. Our tool, starting from virtual machine bytecode, discovers, tracks, and propagates type information which is often implicit in Erlang programs, and reports warnings when a variety of type errors and other software discrepancies are identified. Since the analysis currently starts from bytecode, it is completely automatic and does not rely on any user annotations. Moreover, it is effective in identifying software defects even in cases where source code is not available, and more specifically in legacy software which is often employed in high-reliability systems in operation, such as telecom switches. We have applied our tool to a handful of real-world applications, each consisting of several hundred thousand lines of code, and describe our experiences and the effectiveness of our techniques.

Hello Dialyzer

- **Sound** for defect detection

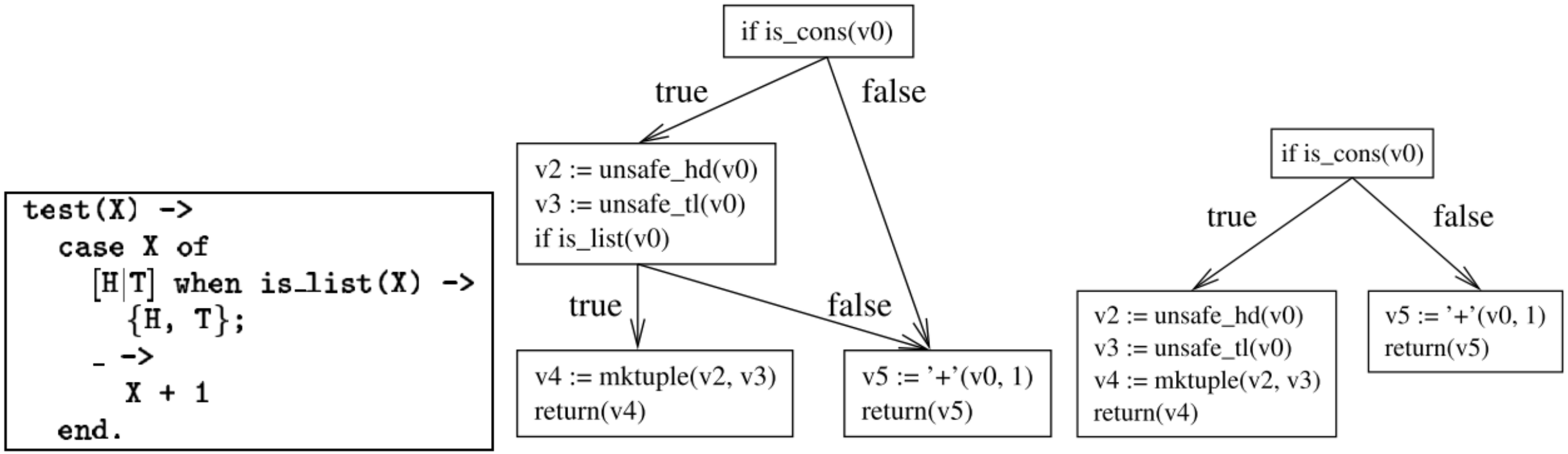
- Never generate FALSE ALARMS (POSITIVES)

- Adapt to Erlang Code Style

- Icode bytecode translation (represented as a CFG)

- Local analysis via PLT (Persistent Lookup Table) for intra-module/cross-module mappings

- *disjoint union of prime types*



(a) ERLANG code (b) Icode w/o optimization (c) Icode w optimization

Fig. 1. ERLANG code with a redundant type guard.

Experience from Developing the Dialyzer: A Static Analysis Tool Detecting Defects in Erlang Applications

Konstantinos Sagonas

Department of Information Technology
Uppsala University, Sweden
kostis@it.uu.se

Abstract

We describe some of our experiences from developing the Dialyzer defect detection tool and overseeing its use in large-scale commercial applications of the telecommunications industry written in Erlang. In particular, we mention design choices that in our opinion have contributed to Dialyzer's acceptance in its user community, things that have so far worked quite well in its setting, the occasional few that have not, and the lessons we learned from interacting with a wide, and often quite diverse, variety of users.

strongly encourages rapid prototyping and performing unit testing early on in the development cycle. Like many functional language implementations, the Erlang/OTP system comes with an interactive shell where Erlang modules can be loaded and the functions in them can easily be tested on an individual basis by simply issuing calls to them. If an exception occurs at any point, it is caught and presented to the user together with a stack trace which shows the sequence of calls leading to the exception. Many errors are eliminated in this way. Of course, testing of multi-thousand (and often million) LOC commercial applications such as e.g. the software of

"laissez-faire style of programming"[9]

—*Konstantinos Sagonas*

TYPER: A Type Annotator of Erlang Code

Tobias Lindahl Konstantinos Sagonas

Department of Information Technology
Uppsala University, Sweden
{tobiasl,kostis}@it.uu.se

Abstract

We describe and document the techniques used in TYPER, a fully automatic type annotator for ERLANG programs based on constraint-based type inference of *success typings* (a notion closely related to principal typings). The inferred typings are fine-grained and the type system currently includes subtyping and subtype polymorphism but not parametric polymorphism. In particular, we describe and illustrate through examples a type inference algorithm tailored to ERLANG's characteristics which is modular, reasonably fast, and appears to scale well in practice.

Categories and Subject Descriptors F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure; D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement—Documentation

General Terms Languages, Theory

Keywords constraint-based type inference, success typings, subtyping, principal typings, Erlang

are that it is completely automatic, never rejects any programs that are accepted by the BEAM compiler, is fast, scalable and reasonably precise, and performs reasonably even when only part of the code base is available.

The rest of the paper is structured as follows. In the next section we briefly review the basis of our work in order to put it into context. The next two sections form the main body of this paper describing TYPER's design goals and basic usage (Section 3) and the type inference algorithm on which TYPER relies in Section 4 which forms the core of this paper. Consequences of inferring success typings for a language with side-effects such as ERLANG are discussed in Section 5. A taste of TYPER's performance appears in Section 6 and the paper ends by reviewing some closely related work and with concluding remarks.

2. The Basis of our Work

2.1 The Erlang language and Erlang/OTP

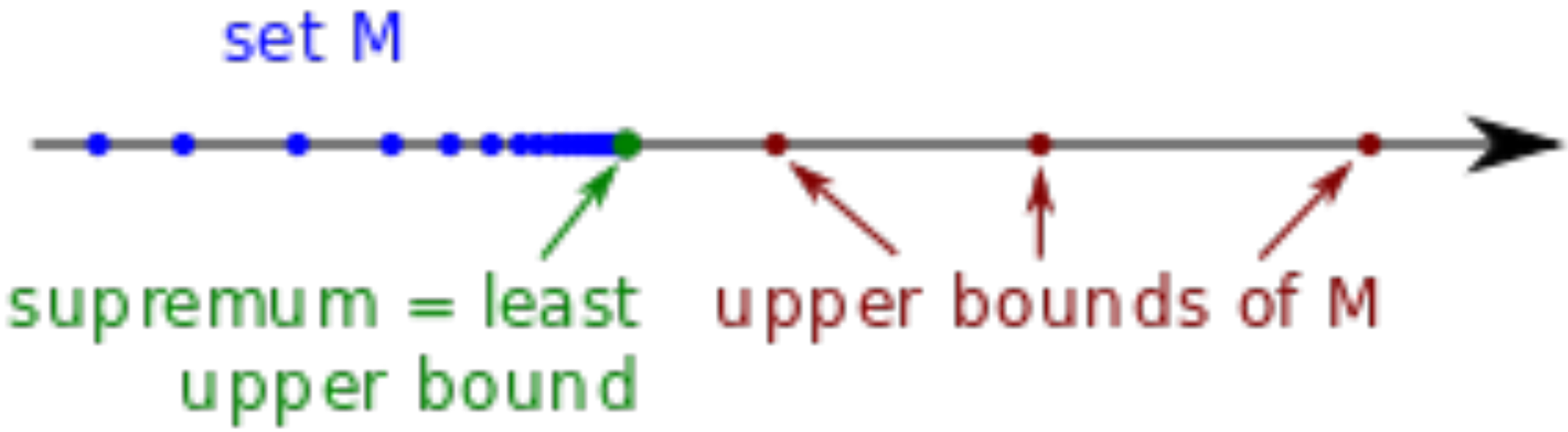
ERLANG [2] is a strict, dynamically typed functional programming

disjoint unions: T_1+T_2 is a "union" of T_1 and T_2 in the sense that its elements include all the elements of T_1 and T_2[25]

- A type is the greatest lower bound of its subtype constraints. To solve a disjunction, all its parts are solved and then the solution is the least upper bound (*sup or supremum*) of the solutions to each disjunctive part.[27]

- $(\tau_x \subseteq 42 \land \tau_{out} \subseteq \text{'true'}) \lor (\tau_{out} \subseteq \text{'false'})$

- $\tau_{out} \subseteq sup(\text{'true'}, \text{'false'}) = \text{bool}()$
  $\tau_x \subseteq sup(42, \text{any}()) = \text{any}()$


set M
supremum = least
upper bound
upper bounds of M

```
%% File: "./and_y.erl"
%% -----------------
-spec andy(_,_) -> boolean().
-spec module_info() -> any().
-spec module_info(_) -> any().

%% File: "./foo.erl"
%% ---------------
-spec length_2([any()]) -> non_neg_integer().
-spec length_3([any()],non_neg_integer()) -> non_neg_integer().
-spec soup(1..10,[atom()]) -> [atom() | integer()].
-spec dejour(_) -> none().
-spec inc(X) -> X when is_subtype(X,number()).
-spec module_info() -> any().
-spec module_info(_) -> any().

%% File: "./hello.erl"
%% -----------------
-spec hello_world() -> 'hello'.
-spec world(pid()) -> 'hi'.
-spec module_info() -> any().
-spec module_info(_) -> any().
```

- Typer Inference is Compositional

- Find most general *success typings* under constraints

- Never rejects programs accepted by BEAM

- Uses forward data-flow analysis to apply a more refined type, using knowledge of call sites

```
-module(m1). -export([main/1]).
main(N) when is integer(N) -> tag(N+42).
tag(N) -> {'tag', N}.

-module(m2). -export([main/1]).
main(N) when is integer(N) -> {tag(N+42), fun tag/1}.
tag(N) -> {'tag', N}.
```

- Use *bottom type* (***none(), but really no_return()***) if conjunction is unsatisfiable (no solution)

Practical Type Inference Based on Success Typings

Tobias Lindahl [1] Konstantinos Sagonas [1,2]

[1] Department of Information Technology, Uppsala University, Sweden
[2] School of Electrical and Computer Engineering, National Technical University of Athens, Greece
{tobiasl,kostis}@it.uu.se

Abstract

In languages where the compiler performs no static type checks, many programs never go wrong, but the intended use of functions and component interfaces is often undocumented or appears only in the form of comments which cannot always be trusted. This often makes program maintenance problematic. We show that it is possible to reconstruct a significant portion of the type information which is implicit in a program, automatically annotate function interfaces, and detect definite type clashes *without* fundamental changes to the philosophy of the language or imposing a type system which unnecessarily rejects perfectly reasonable programs. To do so, we introduce the notion of *success typings* of functions. Unlike most static type systems, success typings incorporate subtyping and never disallow a use of a function that will not result in a type clash during runtime. Unlike most soft typing systems that have previously been proposed, success typings allow for compositional, bottom-up type inference which appears to scale well in practice. Moreover, by taking control-flow into account and exploiting properties of the language such as its module system, success typings can be refined and become accurate and precise. We demonstrate the power and practicality of the approach by applying it to Erlang. We report on our experiences from employing the type inference algorithm, without any guidance, on programs of significant size.

Tried to make it little by little,
tried to make it bit by bit on my own...

and relatively uneventful activity, at least initially. The occasional frustrations of having to convince the type system that one really knows what she is doing are avoided. Also, since type declarations and annotations need not be typed (in), program development can progress more rapidly. Unfortunately, this freedom of expression comes with a price. Significantly less typos and other such mundane programming errors are caught by the compiler. More importantly, the freedom of *not* stating one's intentions explicitly, considerably obstructs program maintenance. In many cases, it is extremely difficult to recall or decipher how a particular piece of code — often written by some other programmer years ago — can be used. Comments are unreliable, often cryptic and confusing, and more often than not rotten. The programmer is much better off if aided by techniques and tools that can help in such situations.

Over the years, many researchers have tried to address such issues. Some have tried to impose and/or tailor a static type system to dynamically typed languages. Despite the technical depth and level of sophistication in many of the proposals, it is fair to say that so far static type systems in dynamically typed languages have enjoyed only limited success in practice. It seems that imposing a static type discipline on a language which was originally designed without one in mind is a Sisyphean task. Other researchers have taken a more low-profile approach and have built useful and successful type inference tools for different programming language paradigms. Among these, we mention soft typing systems [13] and the DrScheme [8] development environment for Scheme, the Ciao

```
%% (list(), any())→any()
length 3([], N) -> N;
length 3([ |T], N) ->
    length 3(T, N+1).
```

Core Erlang

```
'length_2'/1 =
    %% Line 27
    fun (_cor0) ->
     apply 'length_3'/2
         (_cor0, 0)
'length_3'/2 =
    %% Line 29
    fun (_cor1,_cor0) ->
     case <_cor1,_cor0> of
       <[],N> when 'true' ->
           N
       %% Line 30
       <[_cor5|T],N> when 'true' ->
           let <_cor2> =
            call 'erlang':'+'
                (N, 1)
           in  apply 'length_3'/2
                (T, _cor2)
       ( <_cor4,_cor3> when 'true' ->
          ( primop 'match_fail'
                ({'function_clause',_cor4,_cor3})
           -| [{'function_name',{'length_3',2}}] )
        -| ['compiler_generated'] )
end
```

Core Erlang (IR)

$$
\begin{array}{lcl}
e & ::= & X \mid c(e_i, \ldots, e_n) \mid e_1(e_2, \ldots, e_n) \mid f \mid \\
 & & \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid \\
 & & \texttt{letrec}\ x_1 = f_1, \ldots, x_n = f_n\ \texttt{in}\ e \mid \\
 & & \texttt{case}\ e\ \texttt{of}\ (p_1 \rightarrow b_1); \ldots;\ (p_n \rightarrow b_n)\ \texttt{end} \\
f & ::= & \texttt{fun}(x_1, \ldots, x_n) \rightarrow e \\
p & ::= & p'\ \texttt{when}\ g \\
p' & ::= & x \mid c(p'_1, \ldots, p'_n) \\
g & ::= & g_1\ \texttt{and}\ g_2 \mid x_1 = x_2 \mid \texttt{true} \mid \texttt{is\_atom}(x) \mid \\
 & & \texttt{is\_integer}(x) \mid \ldots
\end{array}
$$

"We are instead interested in capturing the biggest set of terms for which we can be sure that type clashes will definitely occur. Instead of keeping track of this set, we will design an algorithm that infers its complement, a function's success typing. ***A success typing is a type signature that over-approximates the set of types for which the function can evaluate to a value."***

— *Lindahl and Sagonas*

"The basic idea is to iteratively solve all constraints in a conjunction until either a fixpoint is reached or the algorithm counters some type clash and fails by assigning the type none() to a type expression.

— Lindahl and Sagonas

- Union Limit + Depth-k abstraction for termination

- Infers success typings for the functions by analyzing its nodes (strongly connected components of the function call graph in a bottom-up fashion)

- Not using conditional or intersection types... so

%% (integer() ∪ list())→integer() ∪ atom()
foo(X) when is integer(X) -> X + 1.
foo(X) -> list to atom(X).

looks like

$$\forall\alpha.(\alpha)\rightarrow(\text{integer}()?(\alpha \cap \text{integer}())) \cup (\text{atom}()?(\alpha \cap \text{list}())) \text{ where } \{\alpha \subseteq \text{integer}() \cup \text{list}()\}$$

A Language for Specifying Type Contracts in Erlang and its Interaction with Success Typings

Miguel Jiménez Tobias Lindahl Konstantinos Sagonas

Department of Information Technology
Uppsala University, Sweden
migueljimg@gmail.com, {tobiasl,kostis}@it.uu.se

1. Introduction

For quite some time now, programs in ERLANG have been developed without any mention of types which describe their intended use. With the advent of automatic documentation tools such as Edoc many ERLANG programmers have discovered the usefulness of types as documentation. However, while type annotations given as comments are better than no annotations at all, they tend to rot as they are not verified. In addition, the usefulness of the type annotations is restricted to the programmer's eyes, and without a standardized type language, tools for static analysis such as Dialyzer cannot take advantage of the information.

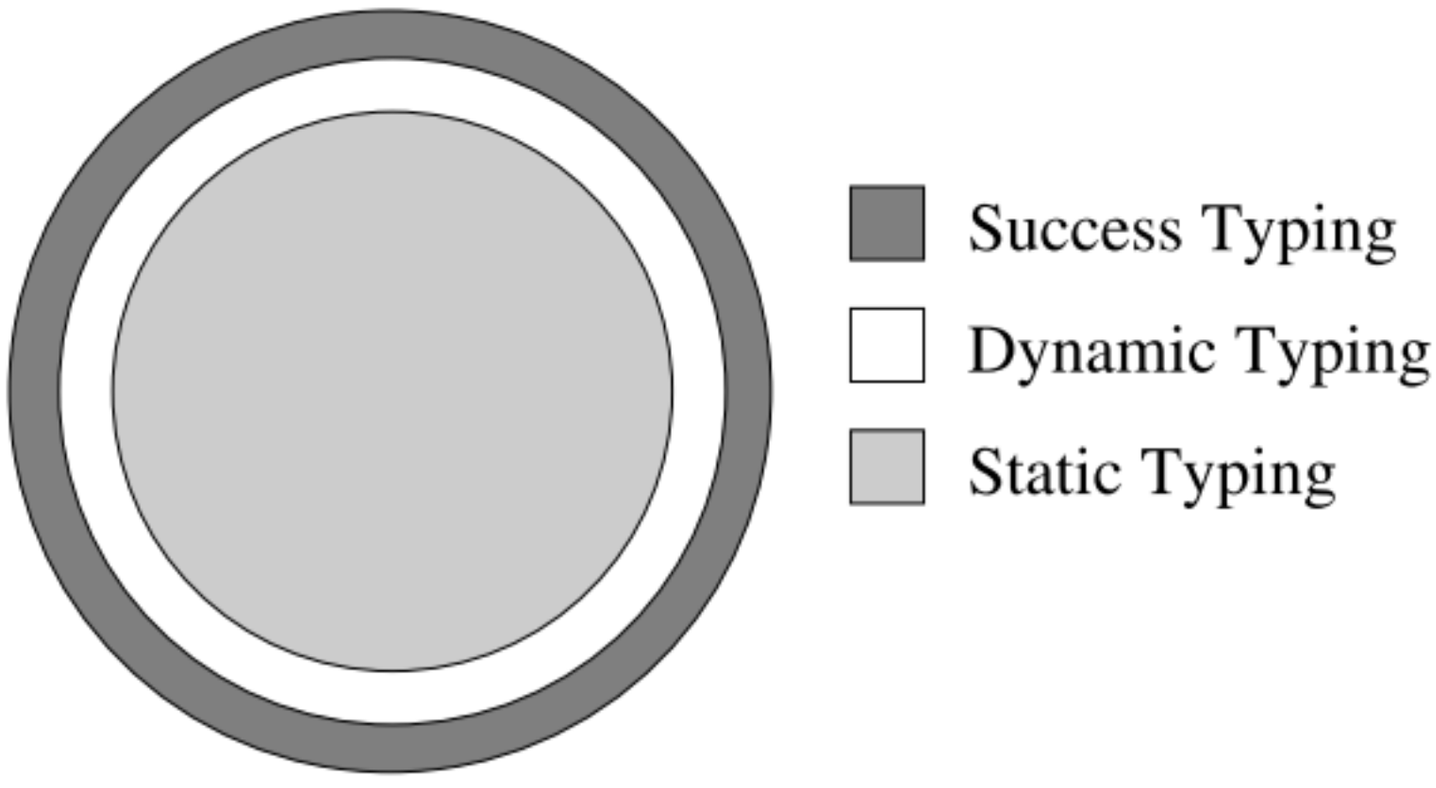
In this work, we propose a contract language that can serve both as documentation in the style of Edoc, and as a guidance to tools such as Dialyzer and TypEr. The contracts are in the form of success typings, a framework developed for expressing type information in dynamically typed programming languages. Our contracts are designed for ease of use and clarity, but also to provide

2. Success Typings

Using type information in dynamically typed languages is often called *soft typing*, a term coined by Cartwright and Fagan [1]. Soft typing encompasses various approaches, but commonly soft type systems use a static type domain extended with some way of expressing dynamic types, either to eliminate dynamic type tests or to find type clashes in the code. Soft type systems are by definition not allowed to reject programs, but they can bring the attention of the user to places in the code where there is a risk for a type clash.

If a soft type system reports all possible points in the code where there is a risk of a type error, we say that the reports (or warnings) are *complete*. If, on the other hand, the soft type system reports only definite type clashes we call the warnings *sound*. With these definitions, the warnings cannot be both sound and complete for a practical programming language, since this is the same problem as having a sound and complete type inference.

In dynamically typed languages type safety is guaranteed by


Success Typing
Dynamic Typing
Static Typing

```
%% is_subtype(X, atom) =:= X :: atom()
-spec inc(X) -> X when is_subtype(X, atom()).
inc(X) when is_integer(X) -> X + 1;
inc(X) when is_float(X) -> X + 1.0.
```

```
typer: Error in contract of function foo:inc/1
    The contract is: (X) -> X when is_subtype(X,atom())
    but the inferred signature is: (number()) -> number()
```

- Contracts allow for more refined analysis/success types

- Function types and polymorphic contracts

```
-spec(all/2 :: (((T) -> bool(), [T]) -> bool())).
```

or

```
-spec id(X) -> X when X :: tuple().
```

- Support for contract overloading

```
-spec(inc/1 :: ((integer()) -> integer());
               ((float()) -> float())).
inc(X) when is integer(X) -> X + 1;
inc(X) when is float(X) -> X + 1.0.
```

Gradual Typing of Erlang Programs: A Wrangler Experience

Konstantinos Sagonas Daniel Luna

School of Electrical and Computer Engineering, National Technical University of Athens, Greece
Department of Information Technology, Uppsala University, Sweden
kostis@cs.ntua.gr daniel.luna@it.uu.se

Abstract

Currently most Erlang programs contain no or very little type information. This sometimes makes them unreliable, hard to use, and difficult to understand and maintain. In this paper we describe our experiences from using static analysis tools to gradually add type information to a medium sized Erlang application that we did not write ourselves: the code base of Wrangler. We carefully document the approach we followed, the exact steps we took, and discuss possible difficulties that one is expected to deal with and the effort which is required in the process. We also show the type of software defects that are typically brought forward, the opportunities for code refactoring and improvement, and the expected benefits from embarking in such a project. We have chosen Wrangler for our experiment because the process is better explained on a code base which is small enough so that the interested reader can retrace its steps, yet large enough to make the experiment quite challenging and the experiences worth writing about. However, we have also done something similar on large parts of Erlang/OTP. The result can partly be seen in the source code of Erlang/OTP R12B-3.

type errors remain in the code. Often these errors appear in the not so commonly executed paths such as those handling serious error situations. Also, type information in the form of comments is often unreliable as it is not checked regularly by the compiler. Such documentation sooner or later is bound to suffer from code rot.

For a number of years now we have been trying to ameliorate this situation by developing and releasing tools that support and promote a different mode of program development in Erlang. Namely, one where most typos, type errors, interface abuses and other software defects are identified automatically using whole program static analysis rather than testing, and where type information is automatically added in the program code, becomes a part of the code, is perhaps manually refined by the programmer and is subsequently automatically checked for validity after program modifications. What's interesting in our approach is that all these are achieved *without* imposing any (restrictive) static type system in the language. Instead, programs can be typed *as gradually as desired* and the programmer has total control of the amount of type information that she wishes to expose and publicly document.

- Testing real projects and exposing type information

- Add explicit type guards in key places in the code.

- Add type declarations and contracts

Compilers as Assistants[21]

```
-- TYPE MISMATCH ------------------------------------------------------ tmp.elm

The argument to function `getFullName` is causing a mismatch.

21|   getFullName
22|>    {
23|>      firstName = "Sam",
24|>      lastName = "Sample",
25|>
26|>      hairColor = "Brown",
27|>      eyeColor = "Brown",
28|>
29|>      address = "1337 Elite st",
30|>      phoneNumber = "867-5309",
31|>      email = "foo@bar.com",
32|>
33|>      pets = 2
34|>    }

Function `getFullName` is expecting the argument to be:

    { …, phoenNumber : … }

But it is:

    { …, phoneNumber : … }

Hint: I compared the record fields and found some potential typos.

    phoenNumber <-> phoneNumber
```

Type Systems as Assistants[21]

```
-- TYPE MISMATCH ------------------------------------------------------ tmp.elm

The argument to function `getFullName` is causing a mismatch.

21|   getFullName
22|>    {
23|>      firstName = "Sam",
24|>      lastName = "Sample",
25|>
26|>      hairColor = "Brown",
27|>      eyeColor = "Brown",
28|>
29|>      address = "1337 Elite st",
30|>      phoneNumber = "867-5309",
31|>      email = "foo@bar.com",
32|>
33|>      pets = 2
34|>    }

Function `getFullName` is expecting the argument to be:

    { …, phoenNumber : … }

But it is:

    { …, phoneNumber : … }

Hint: I compared the record fields and found some potential typos.

    phoenNumber <-> phoneNumber
```

Precise Explanation of Success Typing Errors *

Konstantinos Sagonas
Department of Information Technology
Uppsala University, Sweden
kostis@it.uu.se

Josep Silva Salvador Tamarit
Department of Information Systems and Computation
Universitat Politècnica de València, Spain
{jsilva,stamarit}@dsic.upv.es

Abstract

Nowadays, many dynamic languages come with (some sort of) type inference in order to detect type errors statically. Often, in order not to unnecessarily reject programs which are allowed under a dynamic type discipline, their type inference algorithms are based on non-standard (i.e., not unification based) type inference algorithms. Instead, they employ aggressive forwards and backwards propagation of subtype constraints. Although such analyses are effective in locating actual programming errors, the errors they report are often extremely difficult for programmers to follow and convince themselves of their validity. We have observed this phenomenon in the context of Erlang: for a number of years now its implementation comes with a static analysis tool called Dialyzer which, among other software discrepancies, detects definite type errors (i.e., code points that will result in a runtime error if executed) by inferring success typings. In this work, we extend the analysis that infers success typings, with infrastructure that maintains additional information that can be used to provide precise (i.e., minimal) explanations about the cause of a discrepancy reported by Dialyzer using program slicing. We have implemented the techniques we describe in a publicly available development branch of Dialyzer.

explicit type annotations on expressions, or to provide type signatures for functions. This ability to develop programs rapidly is currently exploited by many modern dynamic languages used in industry but it comes with a cost: many errors that would be caught by a static type system are detected only at runtime, potentially making programs written in dynamic languages less reliable. To ameliorate the situation, many dynamic languages nowadays come with various sorts of type inferencing approaches and tools, aiming to detect most type errors statically rather than dynamically [2, 5, 7]. Such tools allow these languages to combine the expressivity and flexibility of dynamic typing with the robustness of static typing.

Given a program P, we consider the type of an expression as the (possibly infinite) set of values to which this expression can be evaluated using P. Given two expressions e_1, e_2, such that e_1 is a subexpression of e_2, we say that e_1 produces a type error if the type of e_1 is τ_1, the type expected from e_1 by e_2 is τ_2, and $\tau_1 \cap \tau_2 = \varnothing$. Therefore, in this work the term *type error* refers to a specific point in the source code of the program.

Example 1. Assume that the following three Erlang functions appear in a file `ex1.erl` (for the time being ignore the boxes).

```
1  main(A) ->                                  8  f(X) -> X+1.
```

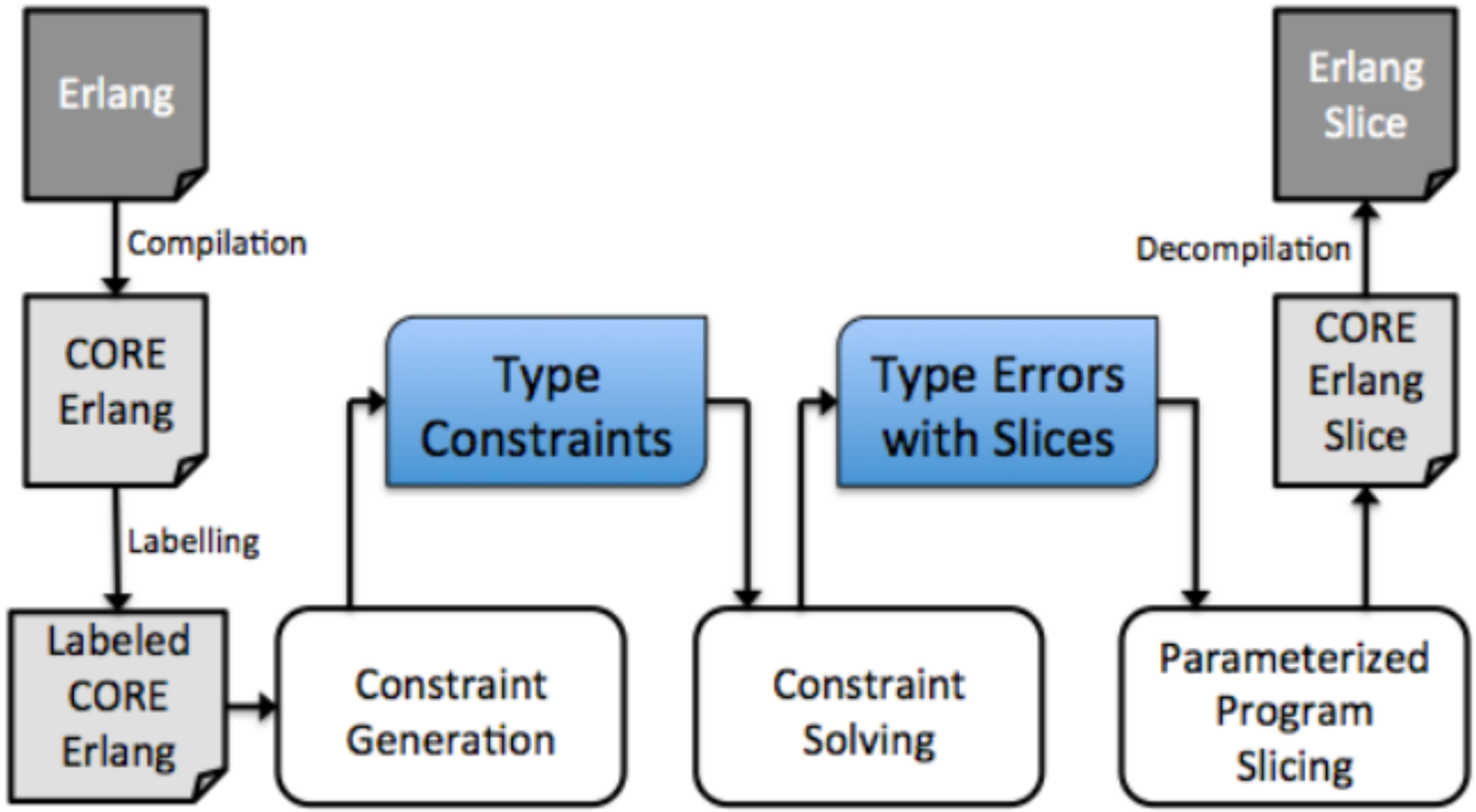
```
main(X) ->
    case X of
        2 ->
            case X of
                1 -> a;
                2 -> b;
                Y -> Y
            end
end.
```

```
> dialyzer --slice
ex3.erl ex3.erl:9: The pattern 1 can never match the type 2
discrepancy sources:
ex3.erl:6 case X of <= Expressions: X
ex3.erl:7 2 -> <= Expressions: 2
ex3.erl:8 case X of <= Expressions: X
ex3.erl:9 1 -> a; <= Expressions: 1

ex3.erl:11: The variable Y can never match since previous clauses
completely covered the type 2 discrepancy sources:
ex3.erl:6 case X of <= Expressions: X
ex3.erl:7 2 -> <= Expressions: 2
ex3.erl:8 case X of <= Expressions: X
ex3.erl:10 2 -> b; <= Expressions: 2
ex3.erl:11 Y -> Y <= Expressions: Y
```

Explaining Success

In computer programming, program slicing is the computation of the set of programs statements, the program slice, that may affect the values at some point of interest, referred to as a slicing criterion. Program slicing can be used in debugging to locate source of errors more easily[].

```
(fun(X^l2) ->
   (case X^l4 of
         (2^l6 when true^l7)^l5 ->
               (case X^l9 of
                     (1^l11 when true^l12)^l10 -> a^l13
                     (2^l15 when true^l16)^l14 -> b^l17
                     (Y^l19 when true^l20)^l18 -> Y^l21
               end)^l8
   end)^l3
end)^l1 .
```


Erlang
Compilation
CORE Erlang
Labelling
Labeled CORE Erlang
Constraint Generation
Type Constraints
Constraint Solving
Type Errors with Slices
Parameterized Program Slicing
Decompilation
CORE Erlang Slice
Erlang Slice

Detection of Asynchronous Message Passing Errors Using Static Analysis

Maria Christakis[1] and Konstantinos Sagonas[1,2]

[1] School of Electrical and Computer Engineering,
National Technical University of Athens, Greece
[2] Department of Information Technology, Uppsala University, Sweden
{mchrista,kostis}@softlab.ntua.gr

Abstract. Concurrent programming is hard and prone to subtle errors. In this paper we present a static analysis that is able to detect some commonly occurring kinds of message passing errors in languages with dynamic process creation and communication based on asynchronous message passing. Our analysis is completely automatic, fast, and strikes a proper balance between soundness and completeness: it is effective in detecting errors and avoids false alarms by computing a close approximation of the interprocess communication topology of programs. We have integrated our analysis in dialyzer, a widely used tool for detecting software defects in Erlang programs, and demonstrate its effectiveness on libraries and applications of considerable size. Despite the fact that these applications have been developed over a long period of time and are reasonably well-tested, our analysis has managed to detect a significant number of previously unknown message passing errors in their code.

-Wrace_conditions***

- Misuse of concurrency primitives can lead to defects around RN, RW, RU, SR.
    - RN: Receive with no messages
    - RW: Receive of the wrong kind
    - RU: Receive with unnecessary patterns (receive w/ never match clauses)
    - SR: Send nowhere received

- Collects pairs of program points possibly involved in a race condition, inspecting every possible execution path, traveling the CFG w/ a depth-first search

- Sharing/alias component to determine if pid refers to the correct process in CFG traversal

- Special care filtering out false alarms

```
-export([start/0]).

start() ->
    Pid = spawn(fun pong/0),
    ping(Pid).

ping(Pid) ->
    Pid ! {self(), ping},
    receive pong -> pang end. %% incorrect false alarm init


pong() ->
    receive {Pid, ping} ->
        Pid ! pong
    end.
```

<tag><c><![CDATA[-Wrace_conditions]]></c>***</tag>
<item>Include warnings for possible race conditions. Note that the analysis that finds data races performs intra-procedural data flow analysis and can sometimes explode in time. **Enable it at your own risk**.
</item>[28]

Erlang is not a strict side-effect-free functional language but a concurrent language[11]

- *Thinking about Concurrency*

- *QuickCheck/PULSE (random scheduling)*

- *Concuerror and Model Checking tools*

III: Playing a Session or N

Session Types

- Session types were designed as a typing discipline for process calculi based on the π-calculus

- Have been called protocols for many years in network and other engineering disciplines which need to treat such patterns.

- Linearity (related to linear logic) is important as channels must not be duplicated, as the duplication of channels will result in the loss of safety guarantees (e.g. must be use exactly once)

Session Typing for a Featherweight Erlang

Dimitris Mostrous and Vasco T. Vasconcelos

LaSIGE, Faculty of Sciences, University of Lisbon

Abstract. As software tends to be increasingly concurrent, the paradigm of message passing is becoming more prominent in computing. The language Erlang offers an intuitive and industry-tested implementation of process-oriented programming, combining pattern-matching with message mailboxes, resulting in concise, elegant programs. However, it lacks a successful static verification mechanism that ensures safety and determinism of communications with respect to well-defined specifications. We present a session typing system for a featherweight Erlang calculus that encompasses the main communication abilities of the language. In this system, structured types are used to govern the interaction of Erlang processes, ensuring that their behaviour is safe with respect to a defined protocol. The expected properties of subject reduction and type safety are established.

Session Typing for a Featherweight Erlang[16]

- Ensure message correlation (*correlation sets*) using unique references via make_ref().

- Operates only over a minimal fragment of Erlang

- Only supports binary sessions, not multiparty ones

Monitoring Erlang/OTP Applications using Multiparty Session Types

Simon Fowler

Monitored Session Erlang[20]

- Erlang's communication patterns are informally defined. How can we apply program logic to guarantee communication safety?

- in MSE, monitors are first class and monitor logic is separated (separate processes) from application/node logic

- The semantics of monitored networks are rejection-based: should a principal attempt to send a message which does not match the specification, the message is not delivered

- Session fidelity proves that safety and transparency hold under reduction.

Scribble

```
module src.com.simonjf.ScribbleExamples.PingPong.PingPong;

global protocol PingPong(role A, role B) {
  rec loop {
    ping() from A to B;
    pong() from B to A;
    continue loop;
  }
}
```

Scribble

```
module src.com.simonjf.ScribbleExamples.PingPong.PingPong_A;

local protocol PingPong at A(role A,role B) {
    rec loop {
        ping() to B;
        pong() from B;
        continue loop;
    }
}
```

Scribble

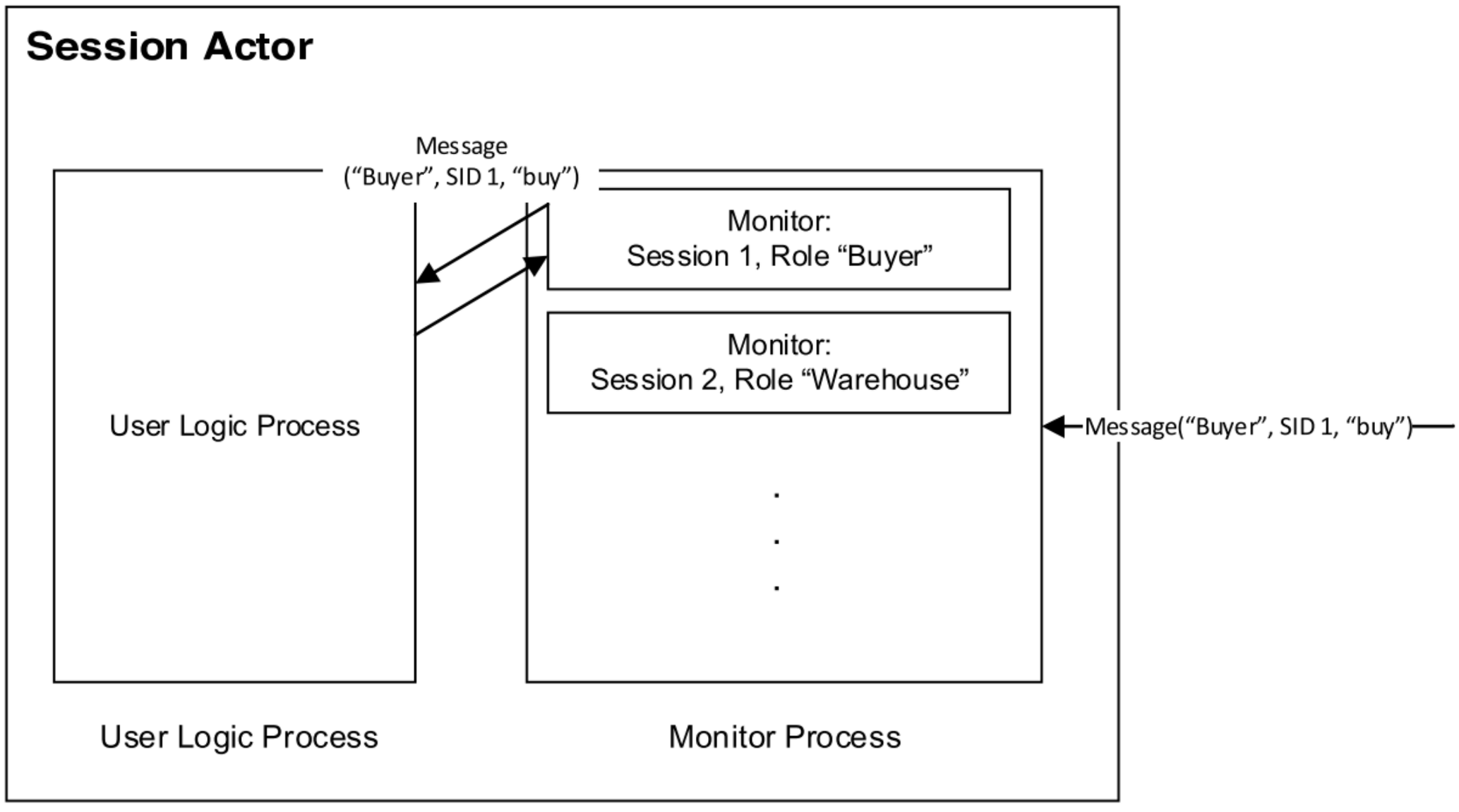
```
module src.com.simonjf.ScribbleExamples.PingPong.PingPong_B;

local protocol PingPong at B(role A,role B) {
    rec loop {
        ping() from A;
        pong() to A;
        continue loop;
    }
}
```

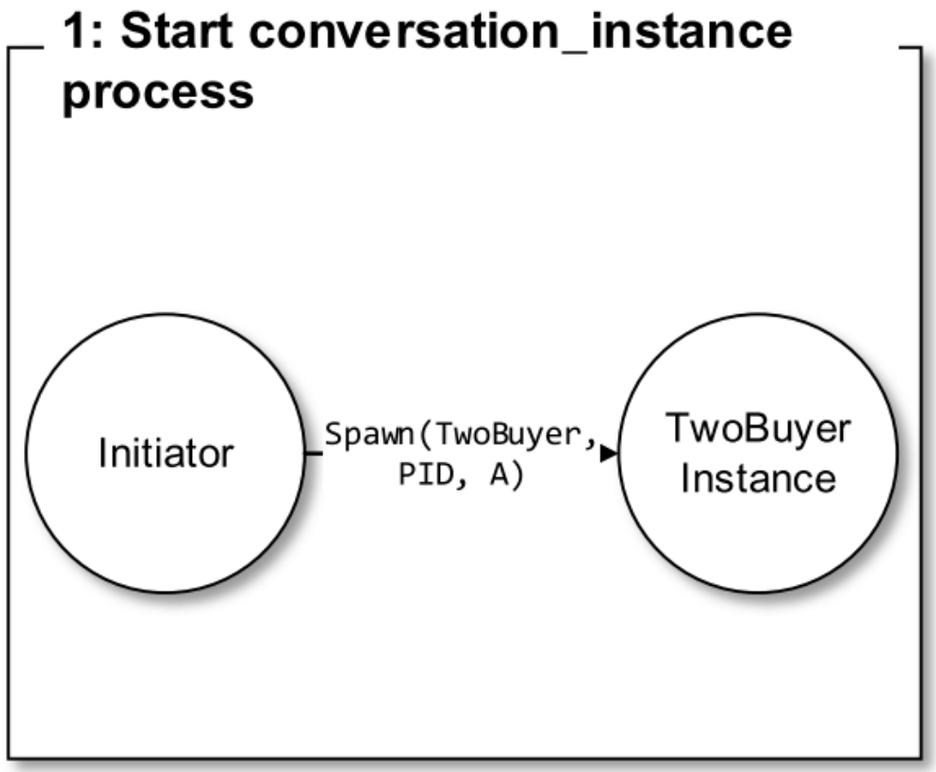
Scribble cont.

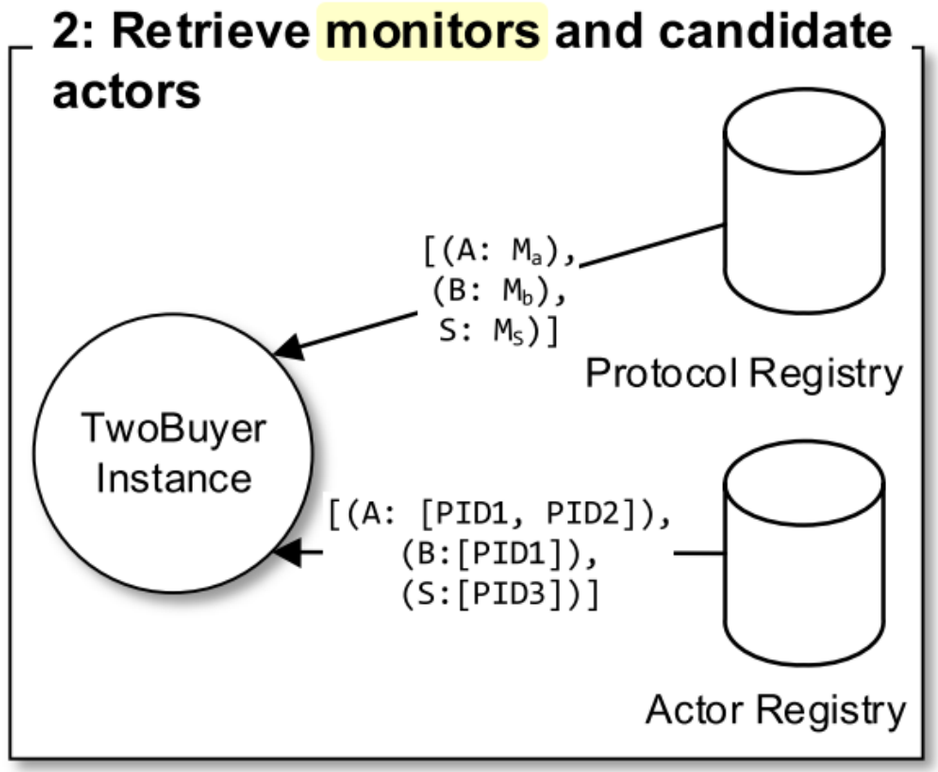
CFSMs

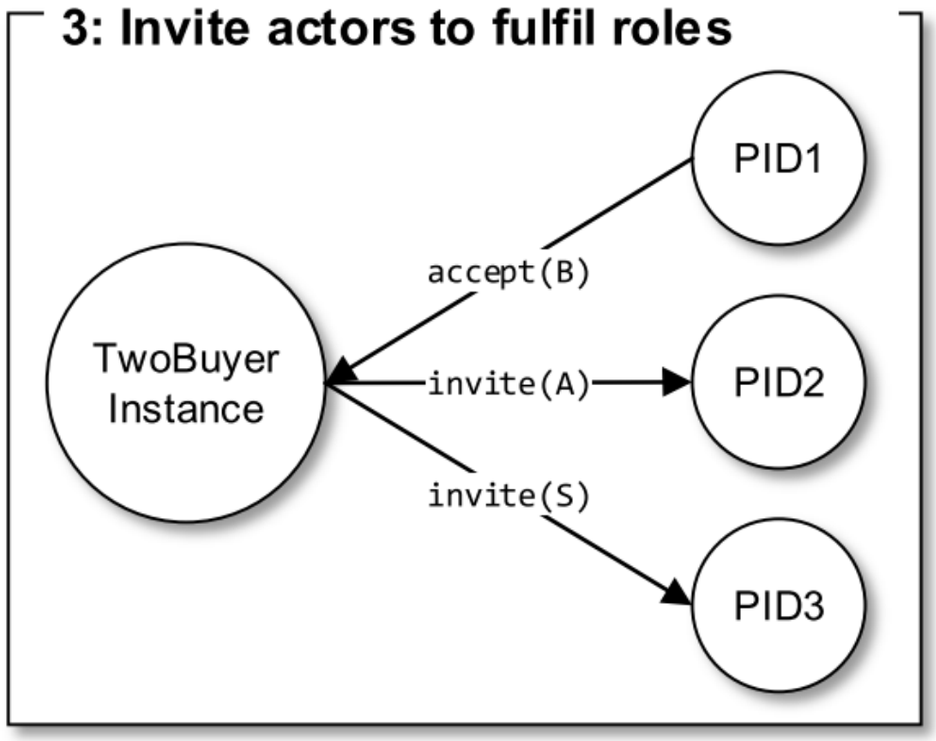
```
module src.com.simonjf.scribbletest.TwoBuyer;

type <erlang> "string" from "" as String;
type <erlang> "integer" from "" as Integer;

global protocol TwoBuyers(role A, role B, role S) {
    title(String) from A to S;
    quote(Integer) from S to A, B;
    // TODO: Loop recursion here
    share(Integer) from A to B;
    choice at B {
        accept(String) from B to A, S;
        date(String) from S to B;
    } or {
        retry() from B to A, S;
        // TODO Loop here
    } or {
        quit() from B to A, S;
    }
}
```

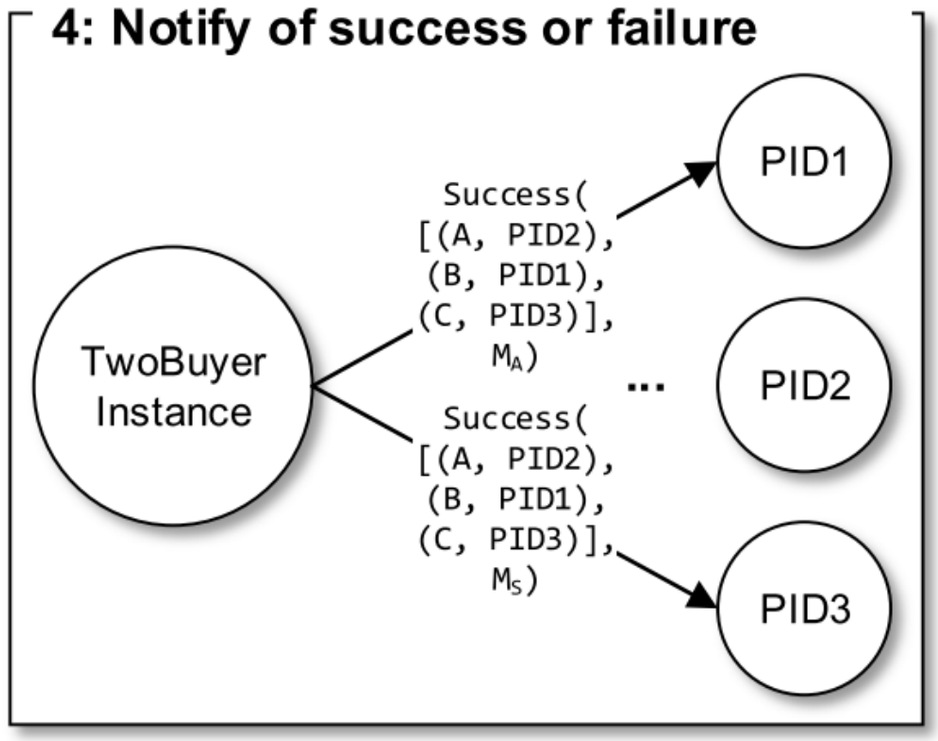
A conversation key is a 3-tuple (M,R,S), where M is the process ID of the monitor, R is the name of the role that the participant is playing in the session, and S is the process ID of the conversation_instance process for the session.

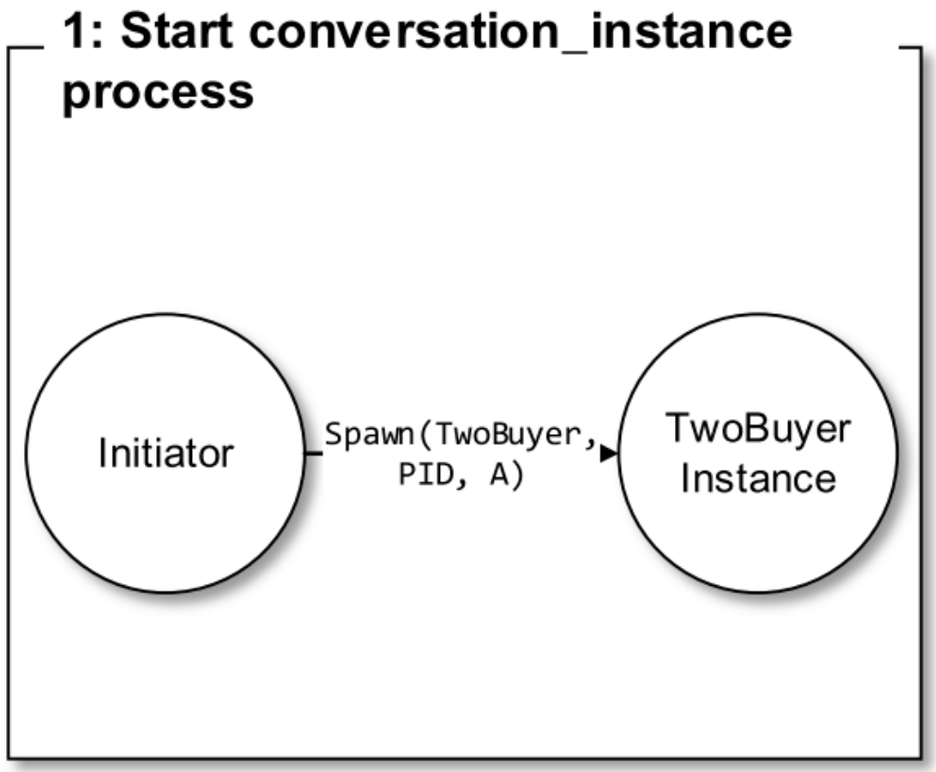


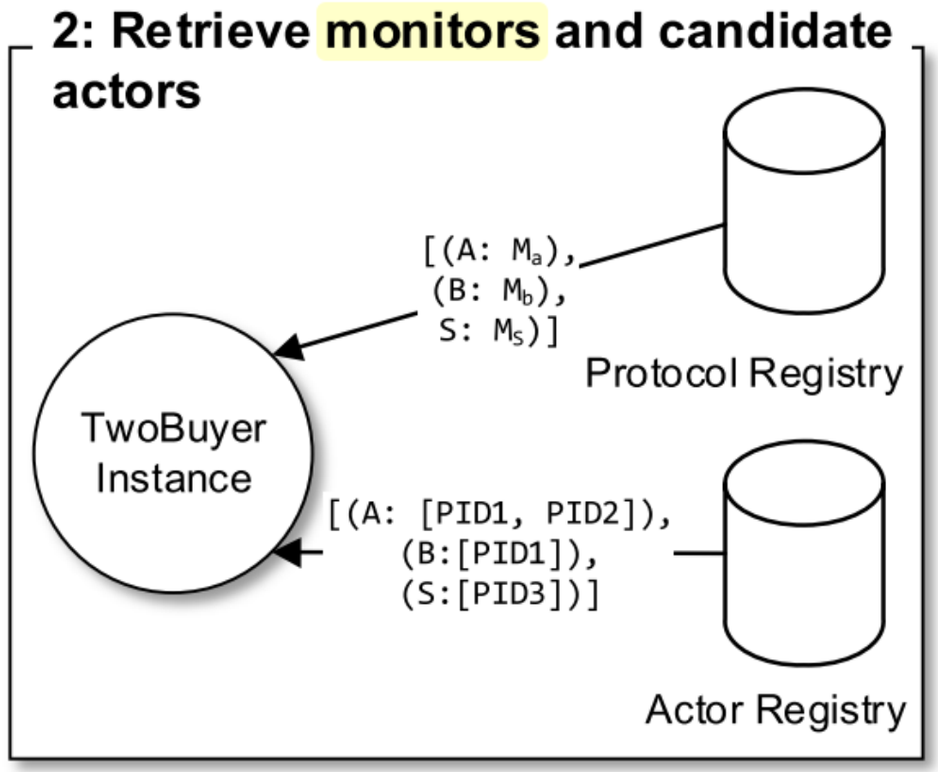

1: Start conversation_instance process
Initiator
Spawn(TwoBuyer, PID, A)
TwoBuyer Instance

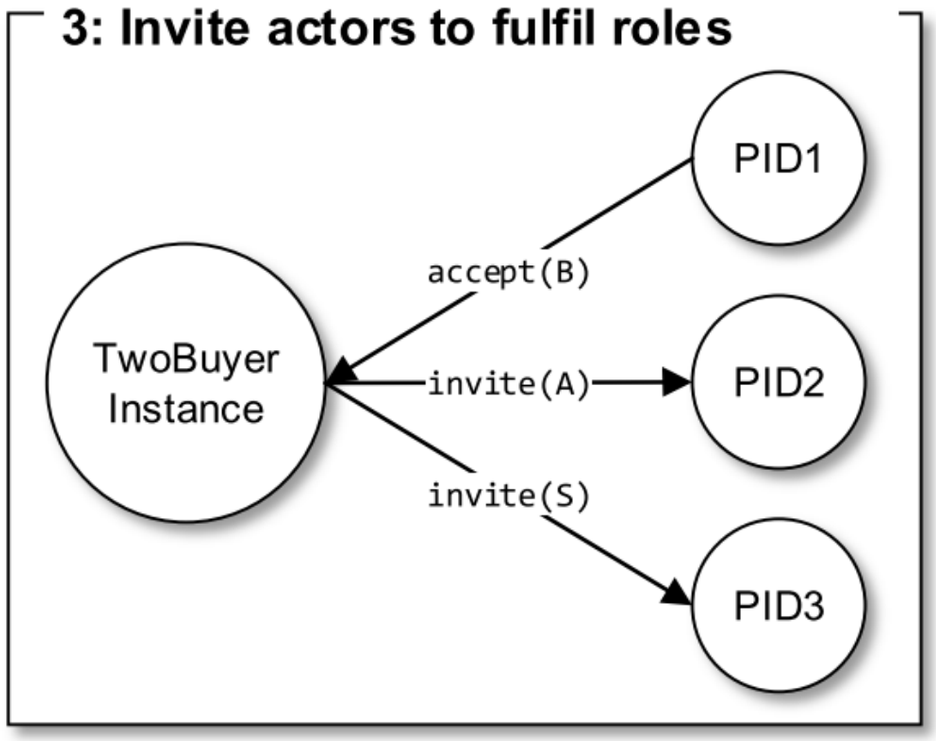

2: Retrieve monitors and candidate actors
[(A: Ma), (B: Mb), S: MS)]
Protocol Registry
TwoBuyer Instance
[(A: [PID1, PID2]), (B:[PID1]), (S:[PID3])]
Actor Registry

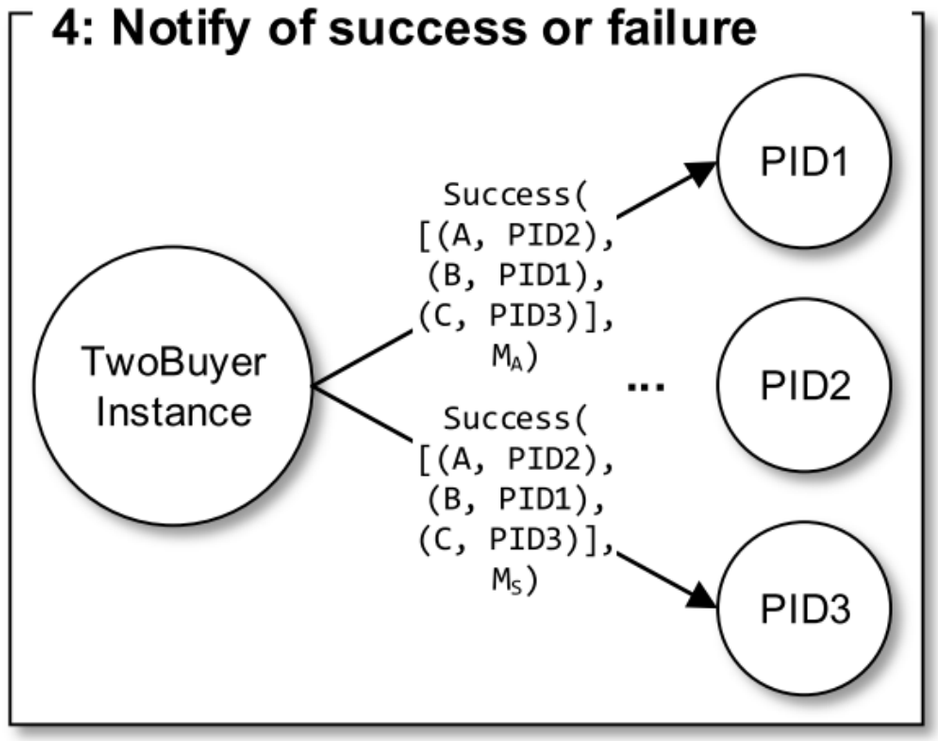

3: Invite actors to fulfil roles
TwoBuyer Instance
accept(B)
invite(A)
invite(S)
PID1
PID2
PID3


4: Notify of success or failure
TwoBuyer Instance
Success([(A, PID2), (B, PID1), (C, PID3)], MA)
Success([(A, PID2), (B, PID1), (C, PID3)], MS)
...
PID1
PID2
PID3


1: Start conversation_instance process
Initiator
Spawn(TwoBuyer, PID, A)
TwoBuyer Instance


2: Retrieve monitors and candidate actors
[(A: Ma), (B: Mb), S: MS)]
Protocol Registry
TwoBuyer Instance
[(A: [PID1, PID2]), (B:[PID1]), (S:[PID3])]
Actor Registry


3: Invite actors to fulfil roles
TwoBuyer Instance
accept(B)
invite(A)
invite(S)
PID1
PID2
PID3


4: Notify of success or failure
TwoBuyer Instance
Success([(A, PID2), (B, PID1), (C, PID3)], MA)
Success([(A, PID2), (B, PID1), (C, PID3)], MS)
...
PID1
PID2
PID3

Thinking Aloud

- what patterns do we see?

- contract/type checks/errors with 3rd-party libs/modules

- session types for epidemic broadcast protocols and *selective hearing*[32]

[1] A. Aiken and B. Murphy, “Static type inference in a dynamically typed language,” Proc. 18th ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang. - POPL ’91, pp. 279–290, 1991.
[2] A. Aiken and E. L. Wimmers, “Type inclusion constraints and type inference,” Proc. Conf. Funct. ..., pp. 31–41, 1993.
[3] Trevor Jim. 1996. What are principal typings and what are they good for?. In Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '96).
[4] A. Lindgren, “A prototype of a soft type system for erlang,” 1996.
[5] S. Marlow and P. Wadler, “A Practical Subtyping System for Erlang,” Int. Conf. Funct. Program., 1997.
[6] A. K. Wright and R. Cartwright, “A practical soft type system for scheme,” ACM Trans. Program. Lang. Syst., vol. 19, no. 1, pp. 87–152, 1997.
[7] S.-O. Nyström, “A soft-typing system for Erlang,” Proc. Erlang Work., pp. 56–71, 2003.
[8] J. Hughes, D. Sands, K. Ostrovsky, “Typing Erlang,” 2002.
[9] K. Sagonas, “Experience from developing the Dialyzer: A static analysis tool detecting defects in Erlang applications,” ... Eval. Softw. Defect Detect. Tools, pp. 1–5, 2005.
[10] T. Lindahl and K. Sagonas, “Practical type inference based on success typings,” Proc. 8th ACM SIGPLAN Symp. Princ. Pract. Declar. Program. - PPDP ’06, p. 167, 2006.
[11] J. Armstrong, “A History of Erlang,” ... Conf. Hist. ..., 2007.
[12] M. Jimenez, T. Lindahl, and K. Sagonas, “A language for specifying type contracts in Erlang and its interaction with success typings,” ... SIGPLAN Work. ERLANG ..., pp. 11–17, 2007.

[13] K. Sagonas and D. Luna, "Gradual typing of Erlang programs: A Wrangler experience," Proc. 7th ACM SIGPLAN Work. Erlang, pp. 73–82, 2008.
[14] M. Christakis and K. Sagonas, "Static Detection of Deadlocks in Erlang," pp. 1–16, 2010.
[15] M. Christakis and K. Sagonas, "Detection of asynchronous message passing errors using static analysis," Proc. 13th Int. Conf. Pract. Asp. Declar. Lang., pp. 5–18, 2011.
[16] D. Mostrous and V. T. Vasconcelos, "Session typing for a featherweight Erlang," Coord. Model. Lang. 2011, pp. 95–109, 2011.
[17] K. Sagonas, J. Silva, and S. Tamarit, "Precise Explanation of Success Typing Errors," Pepm, pp. 33–42, 2013.
[18] M. Christakis, A. Gotovos, and K. Sagonas, "Systematic testing for detecting concurrency errors in Erlang programs," Proc. - IEEE 6th Int. Conf. Softw. Testing, Verif. Validation, ICST 2013, pp. 154–163, 2013.
[19] K. Honda, R. Hu, R. Neykova, T. Chen, P. Deniélou, and N. Yoshida, "Structuring Communication with Session Types," Concurr. Objects Beyond Pap. Dedic. to Akinori Yonezawa Occas. His 65th Birthd., pp. 1–23, 2014.
[20] S. Fowler, "Monitoring Erlang / OTP Applications using Multiparty Session Types," 2015.
[21] E. Czaplicki, "Compilers as Assistants"., http://elm-lang.org/blog/compilers-as-assistants"
[22] D. Spiewak, "What is Hindley-Milner? (and why is it cool?)"., http://www.codecommit.com/blog/scala/what-is-hindley-milner-and-why-is-it-cool
[23] T. Lindahl and K. Sagonas, "TYPER: A Type Annotator of Erlang Code"
[24] B. Harper, "Dynamic Languages are Static Languages"., https://existentialtype.wordpress.com/2011/03/19/dynamic-languages-are-static-languages

[25] B. Pierce, “Types and Programming Languages”
[26] Wikipedia, “Program Slicing”., https://en.wikipedia.org/wiki/Program_slicing
[27] T. Lindahl and K. Sagonas, “Detecting software defects in telecom applications through lightweight static analysis: A war story,” Program. Lang. Syst. Proc., vol. 3302, pp. 91–106, 2004.
[28] https://github.com/erlang/otp/blob/maint/lib/dialyzer/doc/src/dialyzer.xml
[29] J. Siek, et al, “Refined Criteria for Gradual Typing”
[30] F. Hébert, “Learn You Some Erlang for Great Good!”
[31] R. Cartwright and M. Fagan, “Soft typing,” ACM SIGPLAN Not., vol. 39, no. 4, p. 412, 2004.
[32] C. Meiklejohn and P. Van Roy, “Selective Hearing: An Approach to Distributed, Eventually Consistent Edge Computation,” 2015.