What are Success Typings and how do they differ from Type Systems?

Kostis Sagonas

School of Electrical and Computer Engineering

National Technical University of Athens

Greece

Joint work with **Tobias Lindahl** (Uppsala Univerity, Sweden)

What are types good for?

- Document programmers' intentions
  - Can be used to prove properties of programs
- Detect programmer errors
  - Typically, the easy to catch ones such as typos
- Help the compiler generate better code
  - By avoiding runtime overheads

What's wrong with these functions?

```
f(X) → X + 1.
```

```
g(42) → 3.14;
g(foo) → bar.
```

Well-typed programs never go wrong?

```
%% (integer()) → integer()
inv(X) → 1 / X.
```

```
last([X]) → X;
last([_|T]) → last(T).
```

Dynamically typed languages

- Have only one type: term() or any()
- Some primitive functions, however, are only defined on subtypes of this type and their arguments need to be checked at runtime

Dynamic typing & type safety

Type safety is provided by the runtime system

  - All terms are tagged with their type, which is checked in primitive operations

- Primitive types:
  - integers, floats, atoms ('foo', 'true'), …
- Structured types:
  - tuples: {'foo', 42}
  - lists: [1, 2, 3]

Experience with typing Erlang

Dialyzer - a Discrepancy Analyzer of Erlang programs

- Uses a type-based forward data-flow analysis to find errors in Erlang code
- Managed to uncover bugs in large, well-tested applications

Our new goal:

Design a type inference that both can be the basis of Dialyzer's analysis and present type signatures of Erlang functions

Considerations

The inferred type signatures should:

- Be easy to interpret by the programmer
- Never lie: Capture all possible (however unintended) uses of functions

The inference algorithm should:

- Be completely automatic
  - No user annotations
  - No type declarations
- Handle cases where not all code is available
- Be relatively fast

An Erlang implementation of logical and

```
and(true, true) → true;
and(false, _)   → false;
and(_, false)   → false.
```

Erlang program

```
bool() ::=  'true' | 'false'
```

```
> and(true, true).
true
> and(false, true).
false
> and(false, gazonk).
false
> and(3.14, false).
false
```

Trial runs

An Erlang implementation of logical and

```
and(true, true) → true;
and(false, _)   → false;
and(_, false)   → false.
```

Erlang program

```
(bool(), bool()) → bool()
```

HM-type signature

```
> and(true, true).
true
> and(false, true).
false
> and(false, gazonk).
false
> and(3.14, false).
false
```

Trial runs

An Erlang implementation of logical and

```
and(true, true) → true;
and(false, _)   → false;
and(_, false)   → false.
```

Erlang program

```
(any(), 'false') → bool()
```

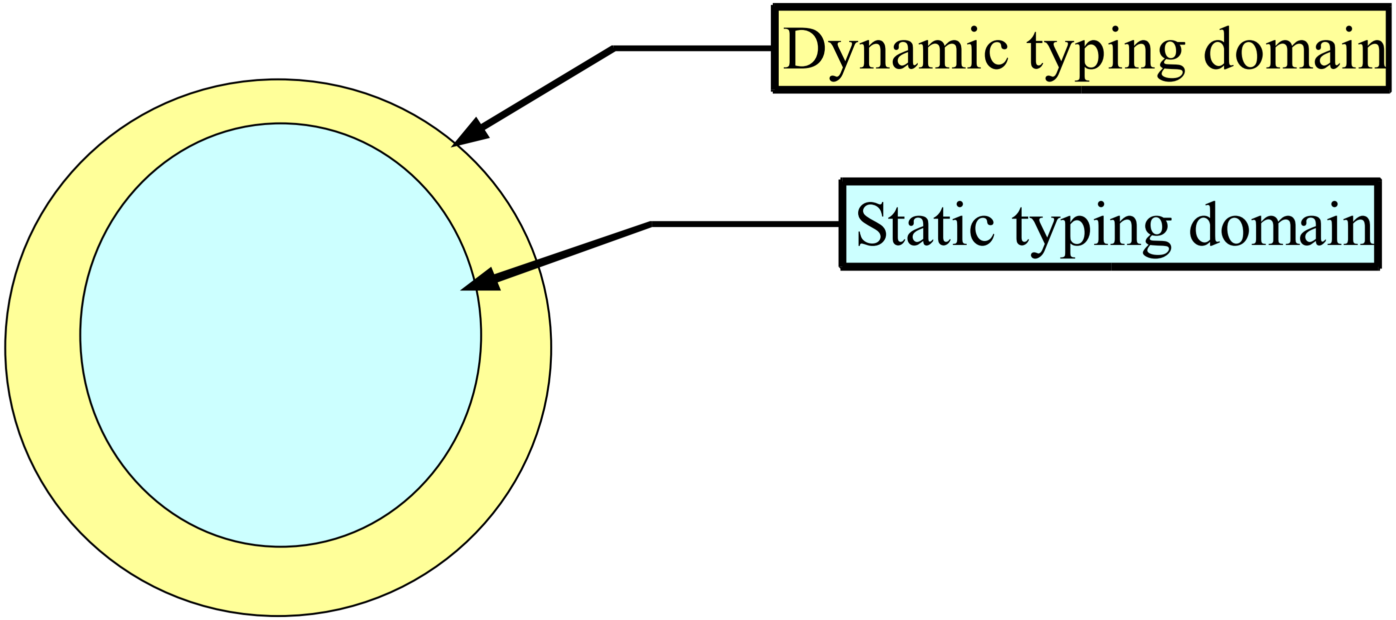
Subtyping signature

```
> and(true, true).
true
> and(false, true).
false
> and(false, gazonk).
false
> and(3.14, false).
false
```

Trial runs

Typing inferred by algorithm from S. Marlow and P. Wadler,
"A practical subtyping system for Erlang"

A quick look at inferred function domains



Dynamic typing domain

Static typing domain

Something needs to be done to capture all of the dynamic range!

Success typings

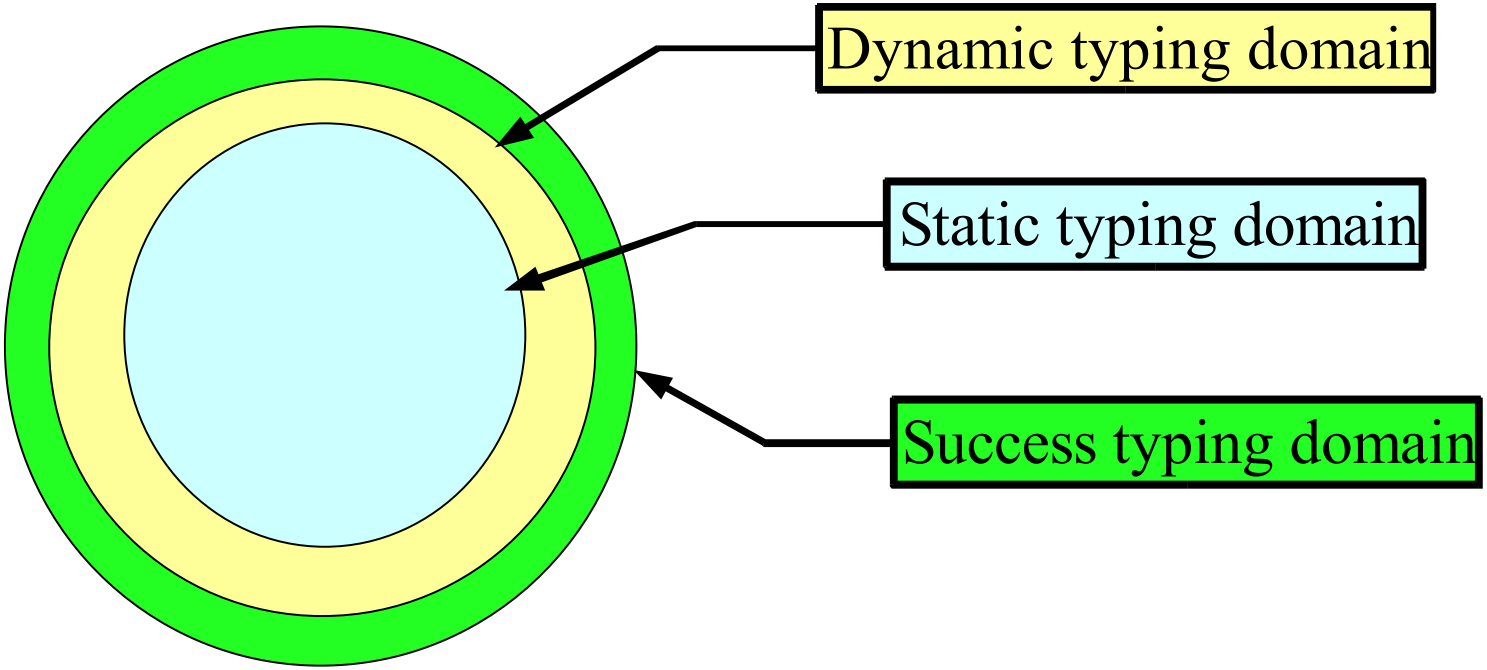
Definition:

A success typing for a function f is a type signature, $(\vec{\alpha}) \to \beta$, such that whenever an application $f(\vec{p})$ reduces to a value v, then $v \in \beta$ and $\vec{p} \in \vec{\alpha}$.

Intuition:

If the arguments are in the domain of the function the application might succeed, but if they do not the application will definitely fail.

Function domains revisited



Dynamic typing domain

Static typing domain

Success typing domain

Success typing, subtyping and HM-types

```
and(true, true) → true;
and(false, _)    → false;
and(_, false)    → false.
```

Erlang program

(bool(), bool()) → bool()

HM-type signature

(any(), 'false') → bool()

Subtyping signature

(any(), any()) → bool()

Success typing

Two sides to the story

Well-typed programs do not go wrong!

Pessimism: If we cannot prove type safety we must reject the program.

Static typing view

Ill-typed programs will surely fail!

Optimism: If we cannot detect a type clash the program might work.

Success typing view

Inferring success typings

There is a most general success typing for all functions of a certain arity

- (any()) → any() for all functions of arity 1
- (any(), any()) → any() for all functions of arity 2
- …

The aim of the inference algorithm is to reduce both the domain and the range of the success typing as much as possible without excluding any valid terms

The inference algorithm

Constraint-based algorithm

- Constraint generation
- Constraint solving, bottom-up per SCC

Constraints are organized in disjunctions and conjunctions of subtype constraints

$$C ::= (T_1 \sqsubseteq T_2) \mid (C_1 \wedge \ldots \wedge C_n) \mid (C_1 \vee \ldots \vee C_n)$$

Conjunctions come from straight-line code and disjunctions come from choices (case statements)

Some examples of inferred typings (1)

```
%% (integer() | float()) → integer() | float()
a(X) → X + 1.
```

```
%% (integer()) → integer()
b(X) when is_integer(X) → X + 1.
```

```
%% (integer()) → 'ok1'
bar(X) →
    case b(X) of
        42 → ok1;
        gazonk → ok2
    end.
```

Some examples of inferred typings (2)

```
%% (integer() | atom()) → integer() | list()
foo(X) when is_integer(X) → X + 1;
foo(X) → atom_to_list(X).
```

```
%% (none()) → none()
gazonk(X) when is_atom(X) → X + 42.
```

Some examples of inferred typings (3)

```
%% (list()) → integer()
length_1([]) → 0;
length_1([_|T]) → length_1(T) + 1.
```

```
%% (list()) → any()
length_2(L) → length_3(L, 0).

%% (list(), any()) → any()
length_3([], N)      → N;
length_3([_|T], N) → length_3(T, N+1).
```

Refined success typings

Definition:

Let f be a function with success typing $(\vec{\alpha}) \to \beta$.
A refined success typing for f is a typing on the form $(\vec{\alpha}') \to \beta'$, such that

- $\vec{\alpha}' \sqsubseteq \vec{\alpha}$ and $\beta' \sqsubseteq \beta$, and
- For all $\vec{p}$ for which the application $f(\vec{p})$ reduces to a value, $f(\vec{p}) \in \beta'$.

Module system to the rescue

- In modern languages the module system cannot be bypassed
  - Code resides in modules
  - Modules have declared interfaces (exported functions)
- Since the module system protects local functions from arbitrary use, we can collect the types of the parameters of all call sites of these functions
- We can use this information to restrict the domains of module-local functions

The list example revisited

```
-module(my_list_utils).
-export([length_2/1]).

%% (list()) → integer()
length_2(L) → length_3(L, 0).

%% (list(), integer()) → integer()
length_3([], N)      → N;
length_3([_|T], N) → length_3(T, N+1).
```

Concluding remarks

Success typings:

- provide an optimistic view on type inference
- will never reject a program that does not have a definite type clash
- capture all possible uses of functions

Current work:

- Investigate trade offs between precision and scalability
- Allow user declarations and annotations