# Mechanization of Erlang's Success Typing: Informal Proof of Reverse Progress

Svetlana Semenova

December 2023

## 1 Introduction

Applications like telecommunications, banking, and gaming require that systems are always operational. However, since crashes are unavoidable in practice, these systems are built in a distributive way: many processes work together, and the failure of one will not collapse the whole. Oftentimes, these distributed systems are written in languages designed for them, such as Erlang, a dynamically typed, functional programming language [1].

For most programming languages, their type systems—sets of rules that restrict what programs are valid in a language—make it impossible for programs to "go wrong" if they pass the type checker. In essence, most languages adhere to the principle that "if a program is well-typed, it will not go wrong." All languages have different definitions of what it means to go wrong, allowing some classes of errors and not others, but they all have some sort of crash prevention within their design.

Unlike most languages, Erlang is designed with a "Let It Crash" coding philosophy. Given Erlang applications generate hundreds, if not thousands, of processes, it is cheaper to let processes crash, communicate that failure to other parts of the system, and restart, rather than recover a broken process or prevent the crash in the first place, as is the standard in most other languages. Crashing becomes the default, rather than an exceptional case.

This philosophy is exemplified by Erlang syntax allowing you to write functions that always crash, no matter what the input is, such as the following:

```
add2(X) when is_list(X) -> X + 2.
```

When `add2` is called on an input, the program will check if the input satisfies the guard—the `is_list(X)` in the expression. If add2 is called with a list, it will satisfy the guard but crash on the addition operation; if it is called with anything other than a list, it will crash from not satisfying any guard. Despite this, this statement is syntactically correct and will compile. In Erlang, syntactically valid programs can, and often do, go wrong.

Researchers, seeing how critical systems are built in Erlang without safety checks, have aimed to add type systems into Erlang to make programs safer—all of which have never been adopted due to this fundamental philosophical difference. No Erlang programmer, who subscribes to the "Let It Crash" philosophy, wants to use a tool that restricts expressiveness and reduces crashes. A soft typing tool, one that gives warning when type clashes could occur rather than halt compilation, has seen wider adoption [2]. And while this was more widely adopted than previous iterations, it gave too many warnings to be useful.

In response, Lindahl et. al. [3] proposed creating a type system where only type clashes that provably always crash are reported and found. The team implemented Dialyzer, a tool that only reports provable type clashes. Dialyzer became an indispensable part of most Erlang workflows, and its popularity inspired the authors to formalize the type system Dialyzer implicitly worked with: success typing.

In a more traditional type system, if a function f has type $(\alpha) \to \beta$, then any value in $(\alpha)$ will reduce to a value in $\beta$ without crashing. This means that if a function cannot be assigned such a type, there exists some value in $(\alpha)$ that will not reduce to a value in $\beta$. Success typing has a converse definition: if a function f has a success typing $(\alpha) \to \beta$, then whenever an input reduces successfully to an output in $\beta$, the input must be of type $(\alpha)$. This states nothing about whether all values of $(\alpha)$ will reduce. If a function has no success typing, that means there is no input that can reduce successfully. In essence, the philosophy of success typing is that "if a program is ill-typed, it necessarily goes wrong." This also means that if a program is well-typed, it has the potential to succeed, even if it can also fail.

Lindahl et. al.'s work enumerated the typing rules and constraint solving algorithms that were implemented in Dialyzer and provided an informal proof of termination of the constraint solving algorithm, though the paper did not address whether any other common properties of type systems hold, such as progress, preservation, type soundness, determinism, and normalizability. As they are typically stated for other type systems, these properties cannot hold in a success typing system—nearly all metatheory that PL researchers typically draw from falls apart in the face of success typing's converse setup.

The project I began this semester aims to accomplish two goals: mechanize and formalize the type system in a proof assistant, specifically Coq, and create a theoretical foundation for it, either by modifying existing property statements or finding new properties. I will be building off Bereczky et. al., which mechanized core-Erlang and mini-Erlang [4]. This paper has a progress update in the form of specifying the Erlang constructs I'll be working with and an informal proof of reverse progress.

# 2 Specification

## 2.1 Specification of Mini-Erlang

This specification of the mini-Erlang is taken and modified from Lindahl et. al.

$$
\begin{aligned}
e ::= & \; x \\
& \mid \text{literals} \\
& \mid (e_1, \ldots, e_n) \\
& \mid \text{fun}(x_1, \ldots, x_n) \to e \\
& \mid \text{let } x = e_1 \text{ in } e_2 \\
& \mid \text{letrec } x = f \text{ in } e \\
& \mid \text{case } e \text{ of } (p_1 \text{ when } g_1 \to b_1); \\
& \ldots; (p_n \text{ when } g_n \to b_n) \text{ end} \\
p ::= & \; x \\
& \mid \text{literals} \\
& \mid (p_1, \ldots, p_n) \\
g ::= & \; g_1 \text{ and } g_2 \\
& \mid x_1 = x_2 \\
& \mid \text{true} \\
& \mid \text{is\_atom}(x) \ldots
\end{aligned}
$$

Terms are composed of variables, literal, tuples, functions, let statements, and case statements. Case statements have patterns and guards; from Lindahl et. al., "a term $t$ matches a pattern $p$ if the variables in $p$ can be bound so that $p$ represents a term syntactically identical to $t$;" and guards are for here restricted to conjunctions of simple constraints on equality and whether variables are of certain literal types.

The most glaring omission of this current set up is the omission of mutual recursion in `letrec`. Adding that into expressions would constitute the replacing the current `letrec` rule with the following:

$$\mid \text{letrec } x_1 = f_1, \ldots x_n = f_n \text{ in } e$$

This would add mutual recursion into the language. However, for the sake of temporary simplicity with the proof-of-concept informal proofs, I am only going to focus on simple recursion, rather than the full mutual recursion.

## 2.2 Specification of Success Types

The success typing system is taken from Lindahl et. al. with some modification.

$$
\begin{aligned}
T ::= & \ V \\
& \mid none() \\
& \mid any() \\
& \mid prim() \\
& \mid (T_1, \ldots, T_n) \\
& \mid (T_1, \ldots, T_n) \to T \\
& \mid T_1 \cup T_2 \\
C ::= & \ T_1 \subseteq T_2 \\
& \mid C_1 \wedge \ldots \wedge C_n \\
& \mid C_1 \vee \ldots \vee C_n
\end{aligned}
$$

The types are mostly standard, with a few notes. $V$ stands for type variables. `any()` acts as a Top type, and `none()` as the emtpy type. The union type is also of note: in Lindahl et. al., any union of any types is allowed, but since unions can become large or even infinite within the algorithms, they imposed a fixed size limit after which the union is widened to a supertype.

One difference between this presentation and Lindahl et. al.'s presentation is the absence of a $T$ *when* $C$ type. In Lindahl et. al., they used this type to force abstractions to actually have an abstraction type without adding in a new type of constraint. However, since this could have been encoded very simply in an alternate way, as I'll do below, without the need to have mutual recursion between $T$ and $C$, I decided to omit it. I do not believe this will have significant impact on the mechanization of this system, since I have already nearly achieved much of this first mechanization with this change.

## 2.3 Specification of Typing Judgements and Constraint Solver

The typing judgements, which slight modification, are taken as follows:

$$\frac{}{A \cup \{x \mapsto \tau\} \vdash x : \tau, \emptyset}(\text{VAR})$$

$$\frac{l \in \tau}{A \vdash l : \tau, \emptyset}(\text{LIT})$$

$$\frac{A \vdash e_1 : \tau_1, C_1 \ldots e_n : \tau_n, C_n}{A \vdash c(e_1, \ldots, e_n) : c(\tau_i, \ldots, \tau_n), C_1 \wedge \ldots \wedge C_n}(\text{STRUCT})$$

$$\frac{A \vdash e_1 : \tau_1, C_1 \quad A \cup \{x \mapsto \tau_1\} \vdash e : \tau_2, C_2}{A \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2, C_1 \wedge C_2}(\text{LET})$$

$$\frac{A \cup \{x \mapsto \tau\} \vdash f : \tau', C_f \quad e : \tau, C_e}{A \vdash \text{letrec } x = f \texttt{ in } e : \tau, C_f \wedge C_e \wedge (\tau' = \tau)} (\text{LETREC})$$

$$\frac{A \cup \{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\} \vdash e : \tau_e, C}{A \vdash \texttt{fun}(x_1, \ldots, x_n) \to e : (\tau_1, \ldots, \tau_n) \to \tau_e, C} (\text{ABS})$$

$$\frac{A \vdash e_1 : \tau_1, C_1 \ldots e_n : \tau_n, C_n}{\begin{array}{r} A \vdash e_1(e_2, \ldots, e_n) : \beta, (\tau_1 = ((\alpha_2, \ldots, \alpha_n) \to \alpha) \wedge (\beta \subseteq \alpha) \wedge \\ (\tau_2 \subseteq \alpha_2) \wedge \ldots \wedge (\tau_n \subseteq \alpha_n) \wedge C_1 \wedge \ldots \wedge C_n \end{array}} (\text{APP})$$

$$\frac{\begin{array}{c} A \cup \{x \mapsto \tau_p | v \in Var(p_1)\} \vdash (p_1 \text{ when } g_1) : \alpha_1, C_1^p \quad b_1 : \beta_1, C_1^q \\ \vdots \\ A \cup \{x \mapsto \tau_p | v \in Var(p_n)\} \vdash (p_n \text{ when } g_n) : \alpha_n, C_n^p \quad b_1 : \beta_1, C_n^q \\ A \vdash e : \tau, C_e \end{array}}{\begin{array}{r} A \vdash \text{case } e \text{ of } (p_1 \text{ when } g_1 \to b_1); \ldots; (p_n \text{ when } g_n \to b_n) \text{ end} \\ : \beta, C_e \wedge (C_1 \vee \ldots \vee C_n) \text{ where} \\ C_i = ((\beta = \beta_i) \wedge (\tau = \alpha_i) \wedge C_i^p \wedge C_i^b) \end{array}} (\text{CASE})$$

The one notable difference between this presentation of the rules and Lindahl et. al.'s is that in ABS, instead of setting the type to $\tau$ and then using the $T$ *when* $C$ construct to ensure it is an abstraction type, I had the rule itself encode it. This is a natural change, especially given the context of the formalization in Coq.

I also added one additional rule: the somewhat vague LIT rule for literals: literals are of a specific type if they are in the set of that type. This rule has not been mechanized, and is currently only present in this paper. I believed it useful to have consideration for literals,

Lindahl et. al. also had one additional rule for patterns, which gets invoked in the CASE rule.

$$\frac{A \vdash p : \tau, C_p \quad A \vdash g : \text{true}, C_g}{A \vdash p \text{ when } g : \tau, C_p \wedge C_g} (\text{pat})$$

This has two slight abuses of notations. First, by stating $A \vdash g : \text{true}, C_g$, Lindahl et. al. meant that $g$, the guard, evaluates to `true` in the given context. This evaluation of guards, however, was not given formally, and assumed to be possible. In the mechanization that I'm working on, then, I created a `guard_eval` function that can evaluate a guard during this type checking process. Second, technically, patterns are not expressions: they're part of the case expressions. So, the addition of this rules, formally, forces the typing relation to accept either patterns or expressions rather than just expressions. The simplest way to get around this is to create a different typing relation only for patterns that gets invoked when typing case expressions, and this is what I am planning on doing.

The typing relation above generates constraints, but the constraint solving itself is also important. Here is the algorithm for constraint solving, taken from Lindahl et. al.

$$\text{solve}(\bot, ) = \bot$$

$$\text{solve}(Sol, \alpha \subseteq \beta) = \begin{cases} Sol & \text{when } Sol(\alpha) \subseteq Sol(\beta) \\ Sol[\alpha \mapsto T] & \text{when } T = Sol(\alpha) \sqcap Sol(\beta) \neq none() \\ \bot & \text{when } T = Sol(\alpha) \sqcap Sol(\beta) none() \end{cases}$$

$$\text{solve}(Sol, Conj) = \begin{cases} Sol & \text{when solve\_conj}(Sol, Conj) = Sol \\ \text{solve}(Sol', Conj) & \text{when } Sol' = \text{solve\_conj}(Sol, Conj) \neq Sol \end{cases}$$

$$\text{solve}(Sol, Disj) = \begin{cases} \bigsqcup Sol' & \text{when } Sol' \neq \emptyset \\ \bot & \text{when } Sol' = \emptyset \end{cases} \text{ where } \begin{cases} Sol' = \{S | S \in PS, S \neq \bot\} \\ PS = \{\text{solve}(Sol, C) | C \in Disj\} \end{cases}$$

$$\text{solve\_conj}(\bot, ) = \bot$$

$$\text{solve\_conj}(Sol, C_1 \wedge \ldots C_n) = \text{solve\_conj}(\text{solve}(Sol, C_1), C_2 \wedge \ldots C_n)$$

$$\text{solve\_conj}(Sol, C) = \text{solve}(Sol, C)$$

This constraint solving algorithm is nearly identical to what is present in Lindahl et. al., except for one typo fixed within the algorithm. A few details of note:

- *Sol* is started as a mapping from all type variables to *any*().
- To solve a conjunction, `solve` will be called on that conjunction until the solution mapping reaches a fixpoint. Lindahl et. al. provided a short proof of termination of this constraint solving algorithm, which I will not recount here.
- The $\sqcap$ symbol is the least upper bound of the two types. $\bigsqcup$ is the greatest lower bound of all types. When applied to a mapping, it is done pointwise.

## 2.4 Operational Semantics: Small-Step

The more common type of operational semantics for Erlang that exist in the literature is small-step. For example, this Lanese et. al. [5] has especially concise small-step semantics that have been referenced often in the literature. However, the Coq formalization of Erlang (Bereczky et. al.) does so through a big-step operational semantics. So, given that the end result will be formalizing success typing in Coq, it would also be beneficial to do these theorems through a big-step lens.

Due to my relative inexperience with big-step operational semantics, especially when it comes to the types of proofs done in CMSC631, I decided to translate the big-step operational semantics in Bereczky et. al. into a small-step operational semantics, using Lanese et. al. as a reference for any other

technicalities. This introduced issues that will become apparent later, and that I will discuss once they appear.

Here are the typeset version of these semantics. Lanese et. al. used $\theta$ as the variable for state, but I will use $\Gamma$, and will use $\hookrightarrow$ to signify stepping. Note that $v$ is used for values only.

$$\frac{\Gamma(x) = v}{\Gamma, x \hookrightarrow \Gamma, v}\text{(S-VAR)}$$

$$\frac{\Gamma, e_i \hookrightarrow \Gamma, e_i'}{\Gamma, (e_1, ..., e_i, ..., e_n) \hookrightarrow \Gamma, (e_1, ..., e_i', ..., e_n)}\text{(S-STRUCT)}$$

$$\frac{\Gamma, e_i \hookrightarrow \Gamma, e_i'}{\Gamma, (e_1, ..., e_i, ..., e_n) \hookrightarrow \Gamma, (e_1, ..., e_i', ..., e_n)}\text{(S-STRUCT)}$$

$$\frac{\Gamma, e_1 \hookrightarrow \Gamma, e_i'}{\Gamma, \text{let } x = e_1 \text{ in } e_2 \hookrightarrow \Gamma, \text{let } x = e_1' \text{ in } e_2}\text{(S-LET1)}$$

$$\frac{}{\Gamma, \text{let } x = v \text{ in } e_2 \hookrightarrow \Gamma \cup \{x \mapsto v\}, e_2}\text{(S-LET2)}$$

$$\frac{\Gamma, f \hookrightarrow \Gamma, f'}{\Gamma, \text{letrec } x = f \text{ in } e \hookrightarrow \Gamma, \text{let } x = f' \text{ in } e}\text{(S-LETREC1)}$$

$$\frac{}{\Gamma, \text{letrec } x = v \text{ in } e \hookrightarrow \Gamma \cup \{x \mapsto v\}, e}\text{(S-LETREC2)}$$

$$\frac{\Gamma, e_1 \hookrightarrow \Gamma, e_1'}{\Gamma, e_1(e_2, \ldots, e_n) \hookrightarrow \Gamma, e_1'(e_2, \ldots, e_n)}\text{(S-APP1)}$$

$$\frac{\Gamma, e_i \hookrightarrow \Gamma, e_i'}{\Gamma, v(e_1, ..., e_i, ..., e_n) \hookrightarrow \Gamma, v(e_1, ..., e_i, ..., e_n)}\text{(S-APP2)}$$

$$\frac{v = \text{fun}(x_1, \ldots, x_n) \to e}{\Gamma, v(v_1, \ldots, v_n) \hookrightarrow \Gamma \cup \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}, e}\text{(S-APP3)}$$

$$\frac{\Gamma, e \hookrightarrow \Gamma, e'}{\begin{array}{r}\Gamma, \text{case } e \text{ of } (p_1 \text{ when } g_1 \to b_1); \ldots; (p_n \text{ when } g_n \to b_n) \text{ end} \\ \hookrightarrow \Gamma, \text{case } e' \text{ of } (p_1 \text{ when } g_1 \to b_1); \ldots; (p_n \text{ when } g_n \to b_n) \text{ end}\end{array}}\text{(S-CASE1)}$$

$$\frac{\begin{array}{c}A, \neg\text{match}(v, p_1) \vee \text{guard-eval}(g_1) = \text{false} \\ \vdots \\ A, \neg\text{match}(v, p_{i-1}) \vee \text{guard-eval}(g_{i-1}) = \text{false} \\ A, \text{match}(v, p_1) \wedge \text{guard-eval}(g_1) = \text{true}\end{array}}{\begin{array}{r}\Gamma, \text{case } v \text{ of } (p_1 \text{ when } g_1 \to b_1); \ldots; (p_n \text{ when } g_n \to b_n) \text{ end} \\ \hookrightarrow \Gamma \cup \text{match-context-extend}(v, p_1), b_i\end{array}}\text{(S-CASE2)}$$

Note for application: Erlang uses eager evaluation. Therefore it makes sense to say that substitution only happens when everything becomes a value.

The other rule worth discussing is S-CASE2. What this rule means intuitively is that once the case expression is matching on a value, it finds the first branch for which the value satisfies the guard (which, by Lindahl et. al.'s definition, means the guard's free variables can be substituted in order to be syntactically identical to $v$, and the guard itself evaluates to true), and enters it, setting the free variables in the pattern to the values computed for it to match. Lindahl et. al. did not precisely write out the functions for matching and evaluating the guard, so I followed suit.

However, this entire definition relies on values. So, what are they defined as?

### 2.5 Values

Lanese et. al., while referencing values in their small-step semantics, did not have a definition of what they defined to be values. In Bereczky et. al., their big-step operational semantics actually evaluated expressions directly to values, but their values were an entirely separate category from syntax. Some values were not expressions at all, namely closures.

So, given that I will mostly be working with a small-step semantics for the purposes of the rest of this paper, I will define values in a way that is most logical for the fragment of the language that I have described.

Values are either

- literals,
- `struct` composed of values, or
- any abstraction.

Saying any abstraction is a value is the same choice made in Programming Language Foundations, what CMSC631 covered. It is also similar to the choice Bereczky et. al. made, which is that any closure is a value. This choice will cause a problem down the line, which I will discuss when it comes up.

## 3 Informal Proof of Reverse Progress

The typical progress property states that if a term is well-typed, then it is either a value or can step. This is guaranteeing that well-typed terms do not crash. However, this is necessarily untrue in success typing: well-typed terms can, and often do, get stuck and crash. I propose a modified progress theorem for success typing: if a term is ill-typed, then it can either take a step or is stuck. The proof for this modified progress property will proceed by induction over the typing derivation rules.

*Proof.* Take an ill-typed term: $A \vdash e : \tau, C$ where given $Sol \vDash C$ and $Sol(\tau) = none()$. By induction on the typing derivation, we aim to prove that $e$ is either stuck or can step. It is equivalent, as well, to show that $e$ can never be a value, since all expressions are either values, stuck, or can step.

- VAR: Since a variable is never a value, we have proven the goal. In fact, most of the cases could, technically, proceed in such a straightforward

fashion. However, as this is not particularly illuminating, I will endeavour to describe more of the logic of why a term will either be stuck or can step, rather than just stating that it is not a value. This is both for a deeper understanding of the system and in case the definitions of "value" changes, which is likely to occur.

The variable typing judgement does not interface with the constraint solving algorithm. So, if a variable is ill-typed, that simply means it must not be within the context, therefore not within the environment, meaning it's a free variable. Since in order to step a variable, you must find it in the environment, a variable being ill-typed means stepping is impossible. Therefore, an ill-typed variable is guaranteed to be stuck.

- LITERAL: All literals have concrete types that are not $none()$. If you do claim to have a literal of type $none()$, then you have essentially populated the empty type, which is nonsense. So, this segment of the proof would be finished by inversion.

- STRUCT: As noted previously, for $Sol$ to give back $none()$ for any type variable, it must be the empty mapping $\bot$ itself. Since the top-level constraint for the `struct` rule is conjunction, the `solve_conj` call must have returned $\bot$, which means that at least one of $C_i$ returned $\bot$ given the current state of $Sol$ at the time. This then means that $e_i$ must be ill-typed given the context, and by the inductive hypothesis, this means that $e_i$ must be either be able to step or is stuck. If $e_i$ can step, then the whole $e$ can step, and our goal is proven. If $e_i$ is stuck, then even if all the rest of the $e_j$'s are values, then entirety of $e$ will either be stuck or can step, proving the goal.

  Note that there is one caveat here that I have glanced over: we are, by the process of `solve_conj`, guaranteed that at least one $C_i$ forced a return of $\bot$ given the current state of $Sol$. However, we are not actually guaranteed that $e_i$ on its own is ill-typed. Perhaps, for example, all $e_j$'s are well typed alone, but when we try to solve the constraints for all of them simultaneously, that's when we run into issues, and $C_i$ is simply the first place where all of the constraints clash against each other.

  I'm still relatively certain that this means that $e$ will not be able to step down to a value. However, I have not been able to construct an example of a `struct` that is ill-typed but all components are well-typed to ground this intuition to reality. As well, this weakness is present in most of the following cases as well, and I'm unsure how things will work out once formalization begins in earnest.

- LET: Similarly to VAR, this segment of the proof could be concluded simply by stating a let expression can never be a value. However, since this is not illuminating, here is a bit more of an exploration of the case of an ill-typed let.

If you have an ill-typed let expression, then that means either $C_1$ or $C_2$ triggered `solve_conj` to return $\bot$. If $C_1$ did, then that means that $e_1$ is ill-typed. By the inductive hypothesis, then, $e_1$ is either stuck or can step. If it is stuck, then the entirety of $e$ is stuck, proving this case; otherwise if it can step, then this case is proven as well. If $C_2$ did, then that means the body of the let expression, with the context extended with the type of $e_1$, is ill-typed, which means that the body will get somehow stuck.

- LETREC proceeds identically to LET due to the similarity in stepping, despite the differences in typing. The one note, though, is that a `letrec` statement can also become ill-typed from it being impossible to assign the recursive $f$ a type. Lindahl et. al. described a separate algorithm to assign types to recursive terms, but did not specify it precisely. Figuring out the algorithm they described implicitly and its details is an important next step.

- ABS: discussion after this proof.

- APP: As usual, more discussion of this case despite not being required.

  An application can step in two ways: either through the stepping of each individual component (the abstraction or any of the inputs) or through substitution of $e_1$'s body with the values of the

  If an application is ill-typed, that means something in the long string of conjunction forced a return of $\bot$ from the constraint solving algorithm. If the first term, the one requiring $e_1$ to be an abstraction, is the one that triggered it, then that means that $e_1$ is not an abstraction, and therefore even if $e_1$ is a value, the entirety of $e$ will not be able to do the substitution step, making $e$ stuck.

  If any of the other $C_i$'s is what forced the return of of $\bot$, then that means either (a) one of the $e_i$'s is ill-typed, with the same caveat as in STRUCT or (b) one of the $e_i$'s type is not a subtype of the input type. If in case (a), then the reduction of each input will not succeed, and the entire $e$ will not be able to enter the substitution step, and so get stuck. If in case (b), then the substitution step will occur, and some operation will get stuck later in computation.

  The same caveat as in STRUCT applies here: perhaps it is the interaction between the many $C_j$ that force a $\bot$ to be triggered. As before, I have not been able to construct an example of such a thing occurring, but will need to investigate this much closer.

- CASE: A case expression can step in one of two ways: either the expression being examined ($e_1$ for this discussion) steps, or, once it is a value, a matching branch is found and stepped to with substitutions of variables in p.

  If a case statement is ill-typed, that means either $C_e$ or $C_1 \vee \ldots C_n$ forced `solve_conj` to return $\bot$. If $C_e$ returned $\bot$, that means that $e_1$ is ill-typed,

and by the inductive hypothesis, $e_1$ can either step or is stuck. If $e_1$ can step, the whole of $e$ can step. If $e_1$ is stuck, then the whole of $e_1$ is stuck, given that entering any branch requires $e_1$ to be a value.

If $C_1 \vee \ldots \vee C_n$ returned $\bot$, that means that every $C_i$ returned $\bot$, meaning that every conjunction $C_i = ((\beta = \beta_i) \wedge (\tau = \alpha_i) \wedge C_i^p \wedge C_i^b)$ had at least one condition that triggered unsatisfiability. This means that for each guard and branch, either:

- The pattern and guard cannot be satisfied given the input $\tau$, or
- The branch is ill-typed.

This then means that no matter which branch execution will try to follow, the program will either get stuck from not matching a guard or from the branch content itself being ill-typed, and by inductive hypothesis, that branch will get stuck.

This part of the informal proof especially illuminates what success typing has "ill-typed" mean: there is no way at all to succeed given something ill-typed.

## 3.1 Discussion On Abstraction

At this point, the elephant in the room is the ABS case. Given how I defined values above, this proof is actually impossible to complete. This is simply because given any abstraction, it is always a value. So, under this configuration, reverse progress is simply false. And this does make sense, in a way: in Erlang, you could have an ill-typed function somewhere in your context, but if you never call it, your program will never get stuck. Your program as a whole is not ill-typed. Even if you entire program is one ill-typed function, nothing will actually crash.

However, we would still like for reverse progress to hold in some form, given how it does a good job at describing the way success typing operates and its philosophy. So, there are a couple of ways that I have considered to get around this:

1. The reverse progress statement could be modified to avoid this issue entirely: simply consider terms that are not abstractions. This will not impact the proof in any way: despite no longer having some sort of inductive hypothesis on any abstraction in any of the steps, given how most cases could have been solved by just showing they could never be a value, this will not cause any issues. I believe that, while this may feel like a tacky workaround, it does actually accurately reflect the way that success typing interacts with evaluation of Erlang programs.

2. If we switched to using the big-step operational semantics, with values being considered an entirely separate category of items, this issue could be avoided. Specifically, under the big-step operational semantics, abstractions are not values, and only closures are. Closures are created by

stepping from an abstractions. So, if the big-step relation required that the body of the abstraction is well typed to step to a closure, then ill-typed abstractions will not step and not be values, therefore proving this leg of the reverse progress proof. Of course, using big-step operational semantics would introduce a load of other problems, but would also solve the need to define the small-step operations myself, as the big-step semantics have already been formally defined in Coq.

# 4 References


1. Erlang. "Erlang: Practical functional programming for a parallel world." Retrieved November 20, 2023. https://www.erlang.org/

2. S. O. Nyström. "A soft-typing system for Erlang." Proceedings of ACM SIGPLAN Erlang Workshop, pages 56-71. ACM 2003.

3. Lindahl, T. and Sagonas, K. "Practical Type Inference Based on Success Typings." ACM 2006.

4. Bereczky, P., Horpácsi, D., and Thompson, S. "A Proof Assistant Based Formalisation of Core Erlang." Trends in Functional Programming. TFP 2020.

5. Lanese, I., Sangiorgi, D., Zavattaro, G. "Playing with Bisimulation in Erlang. Models." Languages, and Tools for Concurrent and Distributed Programming, 2019.