A Falsification View of Success Typing
(extended version)

Robert Jakob and Peter Thiemann

University of Freiburg, Germany
`{jakobro,thiemann}@informatik.uni-freiburg.de`

Abstract. Dynamic languages are praised for their flexibility and expressiveness, but static analysis often yields many false positives and verification is cumbersome for lack of structure. Hence, unit testing is the prevalent incomplete method for validating programs in such languages.

Falsification is an alternative approach that uncovers definite errors in programs. A falsifier computes a set of inputs that definitely crash a program.

Success typing is a type-based approach to document programs in dynamic languages. We demonstrate that success typing is, in fact, an instance of falsification by mapping success (input) types into suitable logic formulae. Output types are represented by recursive types. We prove the correctness of our mapping (which establishes that success typing is falsification) and we report some experiences with a prototype implementation.

1   Introduction

Dynamic languages like JavaScript, Python, and Erlang are increasingly used in application domains where reliability and robustness matters. Their advantages lie in the provision of domain specific libraries, flexibility, and expressiveness, which enables rapid prototyping. However, massive unit testing with all its drawbacks is the primary method of discovering errors: static analysis is often not applicable because it either yields many false positives or restricts the expressiveness. Verification is feasible but cumbersome (see for example the JavaScript formalization effort [3,6]). Moreover, it requires a major effort.

Unit testing with good code coverage is not straightforward to achieve, either. As the development of meaningful unit tests is also cumbersome and time consuming, the lack of static analyses that permit error detection prior to execution is one of the major drawbacks of dynamic languages.

Classical static analyses and type systems guarantee the absence of a particular class of errors: the program cannot *go wrong*. Imposing such a system on a dynamic language deprives it of its major attraction for certain programmers: the ability to write code without being restricted by a formal framework. Even suggesting such a framework would come close to treason. Furthermore,

programmers are confused by false positives or error messages they do not understand [2]. However, an analysis that only reports problems that would definitely lead to an error during execution could be acceptable. This point of view leads to the idea of a success typing.

In a standard type system, the typing $F : \tau_1 \to \tau_2$ means that an application of F to an argument v of type τ_1 yields a result of type τ_2 if $F(v)$ terminates normally. If type checking for the system is decidable, then there are programs which do not lead to type mismatches when executed, but which are rejected by the type system. A trivial example is a conditional that returns values of different types in its branches, but semantically it is clear that only the first branch can ever be executed.

In contrast, a success type system guarantees that for all arguments v **not** of type τ_1, the function application $F(v)$ leads to a run-time error (or nontermination). For an argument v of type τ_1, success typing gives the same guarantees as traditional typing: $F(v) \in \tau_2$ if it terminates normally. By necessity, the guarantee of the run-time error is also an approximation, but success typing must approximate in the other direction as a standard type system. Hence, the "standard part" of a success type usually gives a weaker guarantee than a standard type. In model checking terms, a standard type system performs verification whereas success typing seems related to falsification [1]: its goal is the detection of errors rather than proving the absence of them.

1.1   Success Typings in Erlang

Erlang is a dynamically typed functional programming language with commercial uses in e-commerce, telephony, and instant messaging. Besides the usual numeric and string types, Erlang includes an atom data type for symbols and tuples for building data structures.

Lindahl and Sagonas [10, 14] designed a success typing system for Erlang which infers types with a constraint-based algorithm. Types are drawn from a finite lattice that encompasses types for various atoms (symbols, numbers, strings, etc), functions, tuple and list constructions, unions, and a type *any* that subsumes all other types. One of the major goals of their approach is the ability to automatically generate documentation for functions from the inferred success types. This goal requires small, readable types, which are guaranteed by the finiteness of the lattice. Types for data structures are made finite by cutting off at a certain depth bound. A concrete example shows where this boundedness leads to approximation.

Many Erlang programming idioms rely on named tuples, that is, tuples where the first component is an atom and the remaining components contain associated data as in {book,"Hamlet","Shakespeare"}. One can view named tuples as named constructors: book("Hamlet","Shakespeare"). Named tuples can be nested arbitrarily and created dynamically.

Lindahl and Sagonas' algorithm misses some definite errors based on nested named tuples, as can be seen by the following example. Here is an implementation

of a list length function returning the zero constructor and succ constructor instead of the built-in integers.[1]

```
length([]) -> {zero};
length([_|XS]) -> {succ, length(XS)}.
```

The Dialyzer[2] infers the following success type for length:

$$\text{length} : [any] \rightarrow \text{zero} \cup \text{succ}(\text{zero} \cup \text{succ}(\text{zero}) \cup \text{succ}(any))$$

The argument part of the success type, $[any]$, describes that applying length to a non-list argument yields an error and applying it to a list of arbitrary content might succeed or fail. The result part describes the return value as either zero or as a nested tuple consisting of succ and any value. The argument part is exact: There is no argument of type $[any]$ for which length fails. However, the analyzer restricts tuples to a nesting depth of three levels.

To illustrate the problem with this approximation, consider the function `check` that pattern matches on a nest of named tuples, which cannot be created by the `length` function. Applying the `check` function to the result of `length` yields a definite error. However, the standard setting of the Dialyzer does not detect this error.

```
check({succ,{succ,{succ,{foo}}}}) -> 0.
test() -> check(length([0,0,0,0])).
```

1.2 Our Approach

We focus on errors that include the creation and destruction of data structures and thus consider programs that manipulate constructor trees, only. Our approach describes input type and output type of a function with different models. A success typing of a function comprises a recursive type describing the possible outputs and of a crash condition as a logical formula whose models are the crashing inputs of the function. This approach yields a modular definition of success typings.

Contributions

- We propose a new formally defined view of success typing for a language with data structures. We represent the input and output types of a function differently and thus obtain a modular approach.
- Our approach is correct. We show preservation of types and crash condition during evaluation as well as failure consistency (i.e., if our analysis predicts a crash, the evaluation crashes definitely).
- We give a prototype implementation of our approach.

---

[1] The left-hand side pattern [] matches the empty list and the pattern [_|XS] matches a list with arbitrary head and tail bound to XS.

[2] The DIscrepancy AnalYZer for ERlang programs, an implementation of Lindahl and Sagonas' algorithm. `http://www.erlang.org/doc/man/dialyzer.html`

Outline In Section 2 we define syntax and semantics of a constructor-based language. We introduce types and crash conditions for expressions of this language in Section 3 followed by an analysis that assigns types and crash conditions to expressions. Afterwards, we show show the correctness of the analysis. We discuss practical issues of our approach in Section 4. In Section 5 we discuss related work and conclude in Section 6.

An extended version of this article, including proofs, is available online [8].

2 Language

We illustrate our approach using a higher-order call-by-value language λ_C that comprises of explicit recursion, integer values, n-ary constructors, and pattern-matching to distinguish and destruct the previously defined constructors. We draw these constructors from a fixed, finite, and distinct ranked alphabet, that is, every constructor has a specific arity. We will denote constructors by upper case letters $A, B, C, \ldots$, and implicitly specify their arity when creating constructor terms.
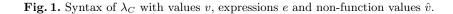
Syntax Syntactically, the language λ_C (Fig. 1) consists of values and expressions. A value v is either an integer literal n, a constructor term $C(v_1, \ldots, v_n)$ where C has arity n and v_i are values, a recursive unary[3] function **rec** f $x = e$, or an explicit error err. An expression e is either a value v, an identifier x, a constructor term $C(e_1, \ldots, e_n)$ where C has arity n and e_i are expressions, a function application $(e\ e)$, or a pattern-matching expression **match** e **with** P. Within the possibly empty list of patterns P, a pattern $C(x_1, \ldots, x_n) \rightarrow e$ consists of a constructor C with arity n, a list of variables $x_1, \ldots, x_n$, and a body expression e. For the list of empty patterns we write $[\ ]$ and to append lists we write $[C(x_1, \ldots, x_n) \rightarrow e] + P$. We assume that the constructors in a list of patterns occur at most once. We introduce an auxiliary definition $\hat{v}$ that represents values not containing functions. For constructor expressions with arity zero, we omit parentheses.

Semantics In Fig. 2 we define the semantics of λ_C as a small-step operational semantics. We use $\mathcal{E}$ to describe expressions with holes $\square$ and EVAL-FINAL to evaluate expressions containing only values as subexpressions. EVAL-HOLE evaluates expressions by choosing holes. SAPP defines recursive function application by capture-avoiding substitution of the argument and function symbol. The rule SMATCH evaluates a constructor value and a list of patterns if the the constructor value matches the first pattern. If so, it extracts the values of the argument and substitutes the variables for the corresponding values in the pattern's body expression. If the first pattern in the list of patterns does not match the constructor value, the rule SMATCHNEXT applies and discards the first non-matching pattern.

---

[3] Multiple arguments can be passed by wrapping them in a constructor.

We explicitly define error creation and propagation as we want to detect definite errors in our programs. Errors occur, if the expression at the first argument of a function application is reduced to a non-function value or if a pattern matching expression occurs with an empty list either because no pattern matched or the list of patterns was initially empty. The former case, non-function values in applications, is handled by the rule SAppErr1 and the latter case by the rule SMatchErr both reducing to the error value err. Error propagation is handled by the rules SMatchErr, if the argument to a pattern matching is an error, the rule SAppErr2 if the argument to a function application is an error, and the rule SMatchNextErr, if a constructor contains an error as a subexpression.

$$v ::= n \mid \textbf{rec } f \ x = e \mid C(v_1, \ldots, v_n) \mid \text{err}$$
$$e ::= v \mid x \mid C(e_1, \ldots, e_n) \mid (e \ e) \mid \textbf{match } e \textbf{ with } \overline{[C_i(\bar{x}) \to e_i]}$$
$$\hat{v} ::= n \mid C(v_1, \ldots, v_n)$$

Fig. 1. Syntax of λ_C with values v, expressions e and non-function values $\hat{v}$.

$$\mathcal{E} ::= \ C(v_1, \ldots, v_n, \square, e_1, \ldots, e_m) \mid \square \ e \mid v \ \square \mid \textbf{match } \square \textbf{ with } [C_i(\bar{x}) \to e_i]$$

$$\frac{\text{EVAL-FINAL}}{e \longrightarrow e'}{e \hookrightarrow e'} \qquad \frac{\text{EVAL-HOLE}}{e \hookrightarrow e'}{\mathcal{E}[e] \hookrightarrow \mathcal{E}[e']}$$

SApp	$((\textbf{rec } f \ x = e) \ v)$	$\longrightarrow e[x \mapsto v, f \mapsto \textbf{rec } f \ x = e]$
SAppErr1	$(\hat{v} \ v)$	$\longrightarrow \text{err}$
SAppErr2	$((\textbf{rec } f \ x = e) \ \text{err})$	$\longrightarrow \text{err}$
SMatch	$\textbf{match } C(\overline{v}) \textbf{ with } [C(\overline{x}) \to e, \ldots]$	$\longrightarrow e[\overline{x_i \mapsto v_i}]$
SMatchErr	$\textbf{match err with } [\ldots]$	$\longrightarrow \text{err}$
SMatchNext	$\textbf{match } C(\overline{v}) \textbf{ with } [D(\overline{x}) \to e] + P$	$\longrightarrow \textbf{match } C(\overline{v}) \textbf{ with } P$
SMatchNextErr	$\textbf{match } C(\overline{v}) \textbf{ with } [\ ]$	$\longrightarrow \text{err}$
SCtorErr	$C(v_1, \ldots, v_n, \text{err}, e_1, \ldots, e_m)$	$\longrightarrow \text{err}$

Fig. 2. Small-step operational semantics for λ_C.

3 Type and Crash Condition

The basic notion of our formalization is a type τ that represents trees created from constructors C on a type level. Furthermore, we represent function values using recursive types. To formalize success types, we represent the possible outputs of a function and the valid inputs of a function differently, thus resulting in a non-standard function type definition where the possible outputs are represented using a type τ and the possible inputs are represented using a crash condition ϕ. Types and crash conditions are defined mutually in Fig. 5. Intuitively, a crash condition for a function is a logical formula whose models are types. These types describe inputs that definitely crash the function.

Types τ comprise of type variables α, an equi-recursive function type written $\mu X.\forall\alpha\,[\phi]\,.\tau$ that includes a type variable α representing the function's argument, a return type τ, and a crash condition ϕ indicating when the function definitely crashes. Furthermore, we define a constructor type τ that captures the types of a constructor expression, a union type $\tau\cup\tau$, an integer type int, and the empty type $\bot$ that has no values. We define two operators that work on types: a type-level function application ($\tau\,@_\tau\,\tau$), and a projection function for constructor types $\tau\!\downarrow_i^C$ that projects the ith component of a type τ if it is a constructor type C. The semantics of these operators is defined in Fig. 6. In our definition, the fix-point formulation μX only occurs together with a function type definition. The type operators are always implicitly applied.

Crash conditions ϕ are defined as atoms true tt and false ff, intersection $\phi\vee\phi$ and conjunction $\phi\wedge\phi$, predicates over types $C\in\tau$ symbolizing that a type τ can be a constructor C, $C\notin\tau$ symbolizing that a type τ is not a constructor type C, and $\forall\notin\tau$ symbolizing that τ is not a function. Furthermore, in Fig. 6 we define an operator ($\tau\,@_\phi\,\tau$) that describes a crash-condition-level function application. Again, the crash condition operator is implicitly applied.

An interpretation $\mathcal{J}$ is a mapping of type variables to types. An interpretation of a type $[\![\tau]\!]_\mathcal{J}$ is a set of types as specified in Fig. 3.

$$
\begin{aligned}
[\![\alpha]\!]_\mathcal{J} &= \{\mathcal{J}(\alpha)\} \\
[\![\mu X.\forall\alpha\,[\phi]\,.\tau]\!]_\mathcal{J} &= \{\mu X.\forall\alpha\,[\phi]\,.\tau' \mid \tau' \in [\![\tau]\!]_{\mathcal{J}'}, \mathcal{J}' = \mathcal{J}\setminus\{\alpha\}\} \\
[\![C(\tau_1,\ldots,\tau_n)]\!]_\mathcal{J} &= \{C(\tau_1',\cdots,\tau_n') \mid \tau_i' \in [\![\tau_i]\!]_\mathcal{J}) \\
[\![\tau_1\cup\tau_2]\!]_\mathcal{J} &= [\![\tau_1]\!]_\mathcal{J} \cup [\![\tau_2]\!]_\mathcal{J} \\
[\![\mathrm{int}]\!]_\mathcal{J} &= \{\mathrm{int}\} \\
[\![\bot]\!]_\mathcal{J} &= \{\}
\end{aligned}
$$

Fig. 3. Definition of an interpretation $\mathcal{J}$ on a type τ.

In Fig. 4 we recursively define an entailment relation $\mathcal{J} \vDash \phi$ for an interpretation $\mathcal{J}$ and a crash condition ϕ.

$$\mathcal{J} \vDash \texttt{tt}$$
$$\mathcal{J} \nvDash \texttt{ff}$$
$$\mathcal{J} \vDash \phi_1 \vee \phi_2 \iff \mathcal{J} \vDash \phi_1 \vee \mathcal{J} \vDash \phi_2$$
$$\mathcal{J} \vDash \phi_1 \wedge \phi_2 \iff \mathcal{J} \vDash \phi_1 \wedge \mathcal{J} \vDash \phi_2$$
$$\mathcal{J} \vDash C \in \tau \iff \exists C(\overline{\tau}) \in [\![\tau]\!]_{\mathcal{J}}$$
$$\mathcal{J} \vDash C \notin \tau \iff \nexists (C(\overline{\tau})) \in [\![\tau]\!]_{\mathcal{J}}$$
$$\mathcal{J} \vDash \forall \notin \tau \iff \nexists (\mu X.\forall \alpha\,[\phi]\,.\tau) \in [\![\tau]\!]_{\mathcal{J}}$$

Fig. 4. Definition of the entailment relation $\mathcal{J} \vDash \phi$.

Example 1. We take the length function of lists as an example using constructors $C_{nil}, C_{zero}, C_{succ}$, and C_{cons} with arities zero, zero, one, and two, respectively.

rec $len\ x =$ **match** x **with** $[C_{nil} \to C_{zero}, C_{cons}(x_1, x_2) \to C_{succ}((len\ x_2))]$

A possible function type for the length function is

$$\tau_{len} = \mu X.\forall \alpha \left[C_{nil} \notin \alpha \wedge \left(\left(C_{cons} \in \alpha \wedge (X\ @_\phi\ \alpha {\downarrow}_2^{C_{cons}}) \right) \vee C_{cons} \notin \alpha \right) \right].$$
$$C_{zero} \cup C_{succ}((X\ @_\tau\ \alpha {\downarrow}_2^{C_{cons}}))$$

whose type is recursively entwined with its crash condition. The derivation of this type is described in Section 3.1. We extract the crash condition that still makes use of τ_{len} via X and get a logical formula with free variable α

$$\phi_{len} = C_{nil} \notin \alpha \wedge \left(\left(C_{cons} \in \alpha \wedge (\tau_f\ @_\phi\ \alpha {\downarrow}_2^{C_{cons}}) \right) \vee C_{cons} \notin \alpha \right)$$

that symbolizes when the function crashes. For example the following interpretation (amongst many others)

$$\mathcal{J} = \{ \alpha \mapsto \bigcup \{ ((\mu X.\forall \alpha\,[\texttt{ff}]\,.C_{zero} \cup C_{cons}\,(\tau, (X\ @_\tau\ \alpha)))\ @_\phi\ C_{unused}) \mid \tau \in \mathcal{T} \}$$

entails the crash condition: $\mathcal{J} \vDash \phi_{len}$. Here, C_{unused} is only needed as a dummy argument to the type-level function. When implicitly applying the type operators, we end up with the infinite type[4]

$$\{ \mu X.C_{zero} \cup C_{cons}(\tau, X) \mid \tau \in \mathcal{T} \}$$

This type represents all lists not ending with a nil but with a zero.

$$\tau ::= \alpha \mid \mu X.\forall \alpha\,[\phi]\,.\tau \mid C(\tau_1,\ldots,\tau_n) \mid \tau \cup \tau \mid \mathtt{int} \mid \bot \mid (\tau\,@_\tau\,\tau) \mid \tau\!\downarrow_i^C$$
$$\phi ::= \mathtt{ff} \mid \mathtt{tt} \mid \phi \vee \phi \mid \phi \wedge \phi \mid C \in \tau \mid C \notin \tau \mid \forall \notin \tau \mid (\tau\,@_\phi\,\tau)$$

Fig. 5. Definition of types τ and crash conditions ϕ.

$$(\tau_1\,@_\tau\,\tau_2) = \begin{cases} \tau_b[\alpha \mapsto \tau_2, X \mapsto \tau_1] & \text{if } \tau_1 = \mu X.\forall \alpha\,[\phi]\,.\tau_b \\ (\tau_1\,@_\tau\,\tau_2) & \text{if } \tau_1 = \alpha \\ (\tau_{11}\,@_\tau\,\tau_2) \cup (\tau_{12}\,@_\tau\,\tau_2) & \text{if } \tau_1 = \tau_{11} \cup \tau_{12} \\ (\tau_1\,@_\tau\,\tau_{21}) \cup (\tau_1\,@_\tau\,\tau_{22}) & \text{if } \tau_2 = \tau_{21} \cup \tau_{22} \\ \bot & \text{otherwise} \end{cases}$$

$$\tau\!\downarrow_i^C = \begin{cases} \tau_i & \text{if } \tau = C(\tau_1,\ldots,\tau_n), 1 \leq i \leq n \\ \tau\!\downarrow_i^C & \text{if } \tau_0 = \alpha \\ \bot & \text{otherwise} \end{cases}$$

$$(\tau_1\,@_\phi\,\tau_2) = \begin{cases} \phi[\alpha \mapsto \tau_2, X \mapsto \tau_1] & \text{if } \tau_1 = \mu X.\forall \alpha\,[\phi]\,.\tau_b \\ (\tau_1\,@_\phi\,\tau_2) & \text{if } \tau_1 = \alpha \\ (\tau_{11}\,@_\phi\,\tau_2) \cup (\tau_{12}\,@_\phi\,\tau_2) & \text{if } \tau_1 = \tau_{11} \cup \tau_{12} \\ (\tau_1\,@_\phi\,\tau_{21}) \cup (\tau_1\,@_\phi\,\tau_{22}) & \text{if } \tau_2 = \tau_{21} \cup \tau_{22} \\ \mathtt{tt} & \text{otherwise} \end{cases}$$

Fig. 6. Type and crash condition operators.

Before introducing the analysis that assigns types and crash conditions to expressions, please note that the question of entailment is not decidable in general.

Lemma 1. *It is undecidable whether for an arbitrary crash condition ϕ there exists an interpretation $\mathcal{J}$ such that $\mathcal{J} \vDash \phi$.*

We discuss possible solutions to this problem in Section 4.

3.1 Analysis

We present our analysis as a type system using a judgment $\Gamma \vdash e : \tau \ \& \ \phi$ that relates a type variable environment Γ, an expression e, a type τ of the expression, and a crash condition ϕ characterizing when the expression crashes. We define the derivation rules in Fig. 7.

The rule T-Rec derives a recursive function type for a recursive function expression by inferring the body's type and crash condition using type variables for the argument and a recursive type formulation for recursive calls. For a function application (T-FunApp) we infer types and crash conditions for both the callee e_1 and the argument e_2. The result type of the function application is the type-level application of the types of the callee and the argument. The function application can crash if either e_1 or e_2 crashes, e_1 is not a function, or the application itself crashes. The latter is symbolized by a crash condition-level function application. The rule T-Identifier derives the type of a variable from the environment and never crashes. An error value err has type $\bot$ and always crashes (T-Error). In rule T-Constructor, a constructor expression has a constructor type with the types of its arguments inferred recursively. A constructor crashes if one of its arguments crashes. Integer literals are handled by T-Integer and always have type int and never crash.

For the pattern matching expression, the type is described by the union of the types of the expression in the patterns. The crash condition is described by the crash condition of the expression to match and the crash conditions of the cases. The crash conditions of the cases are built using an auxiliary judgment: $\phi_m; \tau_0; \Gamma \vdash_p P : \tau_p \ \& \ \phi_p$ where ϕ_m describes the crash conditions accumulated so far, τ_0 describes the type of the expression to match, P the list of patterns which are traversed and τ_p the union of the types of the pattern case's body expression. The type and crash condition of a pattern list is created by two rules: if the pattern list is empty, we return the bottom type and the crash condition accumulated to far. If the pattern list is non-empty, we create the type of the current body expression by binding the variables defined in the pattern and inductively applying the derivation. The current expression can crash, if either the pattern matches ($C \in \tau_0$) and the body expression crashes, or if the pattern does not match at all.

---

[4] For the sake of a simpler type syntax, this type cannot be represented using our type syntax directly. We always have to use type-level applications.

Additionally, we define a subtyping relation $\leq: \tau \times \tau$ in Fig. 8 The relation is standard, except for the rule S-Fun, which requires a logical implication of the crash conditions.

The (output) types derived for an expression are over-approximations whereas the crash conditions describe the possible crashes exactly. The interplay of types and crash conditions ends up with definite errors, because the predicate $C \notin \tau$ describes the question whether it is not possible that the type τ is a constructor C, and similarly for the predicate $\forall \notin \tau$.

T-Rec
$$\frac{\Gamma, x_r : \alpha_r, f_r : X \vdash e : \tau_e \ \& \ \phi_e \qquad \alpha_r \text{ fresh} \quad x_r, f_r \notin \mathbf{dom}(\Gamma)}{\Gamma \vdash \mathbf{rec} \ f_r \ x_r = e : \mu X.\forall \alpha_r \, [\phi_e] \, . \tau_e \ \& \ \mathtt{ff}}$$

T-FunApp
$$\frac{\begin{array}{c} \Gamma \vdash e_1 : \tau_1 \ \& \ \phi_1 \\ \Gamma \vdash e_2 : \tau_2 \ \& \ \phi_2 \end{array}}{\Gamma \vdash (e_1 \ e_2) : (\tau_1 \ @_\tau \tau_2) \ \& \ (\tau_1 \ @_\phi \tau_2) \vee \phi_1 \vee \phi_2 \vee \forall \notin \tau_1}$$

T-Identifier
$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau \ \& \ \mathtt{ff}}$$

T-Error
$$\frac{}{\Gamma \vdash \mathbf{err} : \bot \ \& \ \mathtt{tt}}$$

T-Constructor
$$\frac{\forall i \in \{1, \ldots, n\} : \Gamma \vdash e_i : \tau_i \ \& \ \phi_i}{\Gamma \vdash C(e_1, \ldots, e_n) : C(\tau_1, \ldots, \tau_n) \ \& \ \bigvee \overline{\phi}}$$

T-Pattern-Matching
$$\frac{\begin{array}{c} \mathtt{tt}; \tau_0; \Gamma \vdash_p P : \tau_p \ \& \ \phi_p \\ \Gamma \vdash e_0 : \tau_0 \ \& \ \phi_0 \end{array}}{\Gamma \vdash \mathbf{match} \ e_0 \ \mathbf{with} \ P : \tau_p \ \& \ \phi_0 \vee \phi_p}$$

T-Integer
$$\frac{}{\Gamma \vdash n : \mathrm{int} \ \& \ \mathtt{ff}}$$

T-Pattern-Next
$$\frac{\begin{array}{c} \phi_0 \wedge ((C \in \tau_0 \wedge \phi_e) \vee C \notin \tau_0); \tau_0; \Gamma \vdash_p P : \tau' \ \& \ \phi' \\ \Gamma, x_i : \tau_0 \downarrow_i^C \vdash e : \tau_e \ \& \ \phi_e \quad i = 1, \ldots, n \end{array}}{\phi_0; \tau_0; \Gamma \vdash_p [C(x_1, \ldots, x_n) \to e] + P : \tau' \cup \tau_e \ \& \ \phi'}$$

T-Pattern-Empty
$$\frac{}{\phi_0; \tau_0; \Gamma \vdash_p [\,] : \bot \ \& \ \phi_0}$$

Fig. 7. Derivation rules for the types and crash conditions.

3.2 Properties

To justify our analysis, we prove the preservation of types and crash conditions and the correctness. To do so, we need several auxiliary lemma.

Weakening allows the introduction of a fresh type variable into the type environment without changing anything.

Lemma 2 (Weakening). *For expressions e, types τ, τ_y, and τ_0, an identifier y, conditions ϕ and ϕ_0, and an environment Γ, the following holds:*

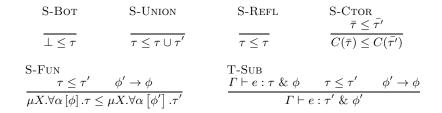1. *If $\Gamma \vdash e : \tau \ \& \ \phi$ and $y \notin \mathbf{dom}(\Gamma)$ then $\Gamma, y : \tau_y \vdash e : \tau \ \& \ \phi$*

$$\begin{array}{cccc}
\text{S-Bot} & \text{S-Union} & \text{S-Refl} & \text{S-Ctor} \\
& & & \dfrac{\bar{\tau} \leq \bar{\tau}'}{} \\[4pt]
\dfrac{}{\bot \leq \tau} & \dfrac{}{\tau \leq \tau \cup \tau'} & \dfrac{}{\tau \leq \tau} & \dfrac{\bar{\tau} \leq \bar{\tau}'}{C(\bar{\tau}) \leq C(\bar{\tau}')}
\end{array}$$

$$\begin{array}{cc}
\text{S-Fun} & \text{T-Sub} \\
\dfrac{\tau \leq \tau' \quad \phi' \to \phi}{\mu X.\forall \alpha\,[\phi]\,.\tau \leq \mu X.\forall \alpha\,[\phi']\,.\tau'} & \dfrac{\Gamma \vdash e : \tau \ \& \ \phi \quad \tau \leq \tau' \quad \phi' \to \phi}{\Gamma \vdash e : \tau' \ \& \ \phi'}
\end{array}$$

Fig. 8. Subtyping rules.

2. If $\phi_0; \tau_0; \Gamma \vdash_p P : \tau \ \& \ \phi$ and $y \notin \mathbf{dom}(\Gamma)$ then $\phi_0; \tau_0; \Gamma, y : \tau_y \vdash_p P : \tau \ \& \ \phi$.

The next lemma shows that we can replace a type variable α within an environment by a concrete type α if we replace all occurrences of the type variable in the resulting type and crash condition. We need this lemma when working with the type-level function application.

Lemma 3 (Consistency of type substitution). *For a well-formed environment Γ, an identifier y, a type variable α, an arbitrary expression e, types τ and τ_α, conditions ϕ and ϕ_0 the following holds:*

1. *If $\Gamma, y : \alpha \vdash e : \tau \ \& \ \phi$ then $(\Gamma, y : \alpha)[\alpha \mapsto \tau_\alpha] \vdash e : \tau[\alpha \mapsto \tau_\alpha] \ \& \ \phi[\alpha \mapsto \tau_\alpha]$*
2. *If $\phi_0; \tau_0; \Gamma, y : \alpha \vdash_p P : \tau \ \& \ \phi$ then $\phi_0[\alpha \mapsto \tau_\alpha]; \tau_0[\alpha \mapsto \tau_\alpha]; (\Gamma, y : \alpha)[\alpha \mapsto \tau_\alpha] \vdash_p P : \tau[\alpha \mapsto \tau_\alpha] \ \& \ \phi[\alpha \mapsto \tau_\alpha]$.*

Item 2 shows that we can substitute a variable y in an expression e with a value of the same type without changing the type and crash condition of the whole expression. This lemma is needed for the type-level function applications later.

Lemma 4 (Consistency of value substitution). *For an environment Γ, an identifier y, types τ and τ_y, an expression e, conditions ϕ and ϕ_0, and a value v, the following holds*

1. *If $\Gamma, y : \tau_y \vdash e : \tau \ \& \ \phi$ and $\Gamma \vdash v : \tau_y \ \& \ \mathtt{ff}$ then $\Gamma \vdash e[y \mapsto v] : \tau \ \& \ \phi$.*
2. *If $\phi_0; \tau_0; \Gamma, y : \tau_y \vdash_p P : \tau \ \& \ \phi$ and $\Gamma \vdash v : \tau_y \ \& \ \mathtt{ff}$ then $\phi_0; \tau_0; \Gamma \vdash_p P[y \mapsto v] : \tau \ \& \ \phi$.*

The next lemma shows that the analysis is designed such that after a successful pattern matching, the crash conditions of the remaining pattern's body expressions cannot be satisfied anymore. The reason is that ϕ_0 in the rules T-Pattern-Next and T-Pattern-Empty influences the resulting crash condition of the whole expression.

Lemma 5 (Unsatisfiability after matching patterns). *For an environment Γ, types τ and τ', and conditions ϕ' it holds that if $\mathtt{ff}, \tau_0; \Gamma \vdash_p P : \tau' \ \& \ \phi'$ then $\nvDash \phi'$.*

Finally, we can establish the preservation theorem for our type system.

Theorem 1 (Preservation of types and crash conditions). *If $\Gamma \vdash e : \tau \& \phi$, $e \hookrightarrow e'$ and $\Gamma \vdash e' : \tau' \& \phi'$ then $\tau \leq \tau'$ and $\phi \leftrightarrow \phi'$.*

Furthermore, we show that our analysis is sound: if the crash conditions report an error, then there is either an error or the evaluation does not terminate.

Theorem 2 (Failure). *If $\forall \mathcal{T}, \mathcal{V}, \Gamma$ and $\forall x \in \mathbf{dom}(\Gamma): \vdash \mathcal{V}(x) : \mathcal{T}(\Gamma(x)) \vDash \mathcal{T}(\phi)$, and*

1. *$\Gamma \vdash e : \tau \& \phi$, then $\mathcal{V}(e) \hookrightarrow^* $ err or $\mathcal{V}(e) \Uparrow$.*
2. *$\phi_0; \tau_0; \Gamma \vdash_p [C_1(\bar{x} \to e_1), \ldots, C_n(\bar{x} \to e_n)] : \tau \& \phi$ and a value $C(v_1, \ldots, v_n)$ with $\Gamma' \vdash C(v_1, \ldots, v_n) : C(\tau_1, \ldots, \tau_n) \& \mathtt{ff}$, then either*
   - *$\forall i : C_i \neq C$*
   - *or $\exists i : C_i = C$ and $(\mathcal{V}'(e_i) \hookrightarrow^* $ err or $\mathcal{V}(e_1) \Uparrow)$.*

4   Practical Considerations

We have shown that success typing is an instance of falsification and thus allows the detection of definite errors. However, as shown by Lemma 1, the satisfiability of ϕ is undecidable in general. Thus a direct algorithmic solution cannot exist. We implemented[5] a version of the analysis that imposes a user-definable limit of k iterations on the unfolding operations described in the operators in Fig. 6 and can thus check for errors up to depth k.

Example 2. An example for a yet problematic combination of type and crash condition we cannot solve at the moment is the following: We create a function that generates an infinite list and apply the resulting stream on the list length function.
  With the list generator's type

$$
\begin{aligned}
\tau_{gen} &= ((\mu X. \forall \alpha \, [\mathtt{ff}] \, . C_{cons}(C_{zero}, (X \, @_\tau \, \alpha))) \, @_\tau \, C_{unused}) \\
&= C_{cons}(C_{zero}, (\tau_{gen} \, @_\tau \, C_{unused}))
\end{aligned}
$$

and the list's type from Example 1 the application of the stream to the length function has the following crash condition after type and crash condition operators are applied once (before substitution):

$$
\left(C_{nil} \notin \alpha \wedge \left(\left(C_{cons} \in \alpha \wedge (X \, @_\phi \, \alpha \downarrow_2^{C_{cons}}) \right) \vee C_{cons} \notin \alpha \right) \right)
$$
$$
[\alpha \mapsto C_{cons}(C_{zero}, (\tau_{gen} \, @_\tau \, C_{unused}))]
$$

After performing the substitution, we can evaluate the predicates that only look finitely deep into their argument. When we apply type and crash condition operators again, we end up on the same crash condition. Although we reach a fix point in this case, this is of course not the case in general.

---

[5] `http://www.informatik.uni-freiburg.de/~jakobro/stpa/`

To solve this problem in general, we need to find an approximation for the crash condition formula. As we only want to find definite errors, our approximation has to be an under-approximation. However, finding a good under-approximation, is yet an open problem.

When we view the output type of a functions as a constructor tree, we can represent it as a higher-order tree grammar, as is proposed by Ong and Ramsey [12]. The (approximated) crash condition of a function can be represented as a tree automaton. As the model checking of tree automata and higher-order tree grammars is decidable [11] we have some means of finding definite errors.

5   Related Work

The idea of finding definite errors in programs is quite old and several approaches exist.

Constraint-based analyses to detect must-information can be found in Reynolds [13] where he describes a construction of recursive set definitions for LISP programs that are "a good fit to the results of a function". However, the goal of the paper was to infer data structure declarations and not to find errors. The constraint-based analysis of Lindahl and Sagonas' [10] is a modular approach similar to ours, but does not account for data structures of arbitrary depth but instead uses k-depth abstraction as we do in our current implementation. Furthermore, the approach of Lindahl and Sagonas uses union types that are widened after a fixed size limit. These limits are to establish small and readable types whereas we focus on exact tracking of values.

Soft typing, presented by Cartwright and Fagan [4] detects suspicious expressions in a program, i.e., expressions that cannot be verified to be error-free, and adds run-time checks. Although the idea of not rejecting working programs is the same, our approach requires no changes in existing programs as we only assume programs to contain errors if we can proof it.

The line of work of Vaziri et al. [5, 7] focuses on imperative first-order languages and uses user-defined specifications given in the Alloy language to state the intention of a function and then checks the implementation against its specification. Although they explicitly mention unbounded data structures in their approach, only instances up to a number of heap cells and loop iterations are considered. In contrast to our approach, they require user-defined annotations. A similar framework [15] removes the chore to define annotations and only requires the user to provide a property to be checked. Their abstraction refines specifications that describe the behavior of procedures and thus creates a refinement-based approach that ensures that no spurious errors appear if the analysis halts.

Different approaches for definite error detections are presented by Ball et al. [1] and Kroening and Weissenbacher [9] for imperative first-order languages in a Hoare-style way. However, a comparison to our approach is difficult because they rely on a transition system to model the behavior of programs whereas we use a type system.

6   Conclusion

We presented a new formal approach to success typings for a constructor-based higher-order language using different representations for the input and output type of a function. We proved that our formulation of success typings is a falsification in the sense that it only reports definite errors. We presented a prototype implementation that checks for errors up to a user-defined bound.

In future we want to look at means to model check (type) trees [11, 12] with logical formula represented as higher-order tree grammars and tree automata, respectively. Thus, we hope to (partly) remove the n-bound of current approaches.
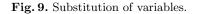
References

1. T. Ball, O. Kupferman, and G. Yorsh. Abstraction for falsification. In K. Etessami and S. K. Rajamani, editors, *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, volume 3576 of *Lecture Notes in Computer Science*, pages 67–81. Springer, 2005.
2. A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Gros, A. Kamsky, S. McPeak, and D. R. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, 2010.
3. M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, and G. Smith. A trusted mechanised JavaSript specification. In S. Jagannathan and P. Sewell, editors, *POPL*, pages 87–100. ACM, 2014.
4. R. Cartwright and M. Fagan. Soft typing. In D. S. Wise, editor, *Proceedings of the ACM SIGPLAN'91 PLDI, Toronto, Ontario, Canada, June 26-28, 1991*, pages 278–292, 1991.
5. J. Dolby, M. Vaziri, and F. Tip. Finding bugs efficiently with a SAT solver. In I. Crnkovic and A. Bertolino, editors, *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*, pages 195–204. ACM, 2007.
6. P. Gardner, S. Maffeis, and G. D. Smith. Towards a program logic for JavaScript. In J. Field and M. Hicks, editors, *Proc. 39th ACM Symp. POPL*, pages 31–44, Philadelphia, USA, Jan. 2012. ACM Press.
7. D. Jackson and M. Vaziri. Finding bugs with a constraint solver. In *ISSTA*, pages 14–25, 2000.
8. R. Jakob and P. Thiemann. A falsification view of success typings. *CoRR*, abs/1502.01278, 2015. extended version.
9. D. Kroening and G. Weissenbacher. Verification and falsification of programs with loops using predicate abstraction. *Formal Asp. Comput.*, 22(2):105–128, 2010.
10. T. Lindahl and K. F. Sagonas. Practical type inference based on success typings. In A. Bossi and M. J. Maher, editors, *PPDP*, pages 167–178. ACM, 2006.
11. C.-H. L. Ong. On model-checking trees generated by higher-order recursion schemes. In *LICS*, pages 81–90. IEEE Computer Society, 2006.

12. C.-H. L. Ong and S. J. Ramsay. Verifying higher-order functional programs with pattern-matching algebraic data types. In T. Ball and M. Sagiv, editors, *POPL*, pages 587–598, Austin, TX, USA, Jan. 2011. ACM Press.

13. J. C. Reynolds. Automatic computation of data set definitions. In *IFIP Congress (1)*, pages 456–461, 1968.

14. K. F. Sagonas, J. Silva, and S. Tamarit. Precise explanation of success typing errors. In E. Albert and S.-C. Mu, editors, *PEPM*, pages 33–42. ACM, 2013.

15. M. Taghdiri. Inferring specifications to detect errors in code. In *19th IEEE International Conference on Automated Software Engineering (ASE 2004), 20-25 September 2004, Linz, Austria*, pages 144–153. IEEE Computer Society, 2004.

A Proofs

For the sake of completeness we enlist the substitution of variables and of type variables in Figs. 9 and 10, respectively.

$$x[x \mapsto v] = v$$
$$y[x \mapsto v] = y$$
$$n[x \mapsto v] = n$$
$$C(\bar{e})[x \mapsto v] = C(\bar{e}[x \mapsto v])$$
$$((e\ e))[x \mapsto v] = (e[x \mapsto v]\ e[x \mapsto v])$$
$$([C_i(\bar{y}) \to e])[x \mapsto v] = [C_i(\bar{y}) \to e[x \mapsto v]] \qquad\qquad x \notin \{\bar{y}\}$$
$$\mathbf{rec}\ f\ y = e[x \mapsto v] = \mathbf{rec}\ f\ y = e[x \mapsto v] \qquad\qquad x \notin \{f, y\}$$
$$\mathbf{rec}\ f\ y = e[x \mapsto v] = \mathbf{rec}\ f\ y = e \qquad\qquad x \in \{f, y\}$$

Fig. 9. Substitution of variables.

Proof (Lemma 1). As types and crash conditions contain type-level functions, applications, and constructors, the crash conditions are Turing-complete and thus satisfiability is not decidable.

Proof (Item 2). Proof by induction on the derivation of the analysis.

 Case T-INTEGER: Let $\Gamma' = \Gamma, y : \tau_y$. Then, $\Gamma' \vdash n :$ int & $\mathtt{ff}$ trivially holds by T-INTEGER.

 Case T-IDENTIFIER: We assume $\Gamma \vdash x : \tau$ & $\mathtt{ff}$ and $y \notin \mathbf{dom}(\Gamma)$. Then, we have two subcases:

   Subcase $x = y$: By inversion, we get $\Gamma(x) = \Gamma(y) = \tau$. Thus, $y \in \mathbf{dom}(\Gamma)$ which contradicts our assumption.

   Subcase $x \neq y$: By inversion, we get $\Gamma(x) = \tau$. Let $\Gamma' = \Gamma, y : \tau_y$. Then, $\Gamma'(x) = \tau$ still holds and we apply T-Identifier and conclude $\Gamma' \vdash x : \tau$ & $\mathtt{ff}$.

 Case T-CONSTRUCTOR: By assumption we have both

$$\Gamma \vdash C(\bar{e}) : C(\bar{\tau})\ \&\ \vee \bar{\phi}$$
$$y \notin \mathbf{dom}(\Gamma)$$

Using inversion we get

$$\Gamma \vdash e_i : \tau_i\ \&\ \phi_i$$

for $i = 1, \ldots, n$. On each of these judgements, we can apply the induction hypothesis, and deduce

$$\Gamma, y : \tau_y \vdash e_i : \tau_i\ \&\ \phi_i$$

$$\alpha[\alpha \mapsto \tau] = \tau$$
$$\beta[\alpha \mapsto \tau] = \beta$$
$$\bot[\alpha \mapsto \tau] = \bot$$
$$C(\bar{\tau})[\alpha \mapsto \tau] = C(\overline{\tau[\alpha \mapsto \tau]})$$
$$(\tau_1 \cup \tau_2)[\alpha \mapsto \tau] = \tau_1[\alpha \mapsto \tau] \cup \tau_2[\alpha \mapsto \tau]$$
$$\mu X.\forall \alpha\,[\phi]\,.\tau[\alpha \mapsto \tau] = \mu X.\forall \alpha\,[\phi]\,.\tau$$
$$\mu X.\forall \beta\,[\phi]\,.\tau[\alpha \mapsto \tau] = \mu X.\forall \beta\,[\phi[\alpha \mapsto \tau]]\,.\tau[\alpha \mapsto \tau]$$
$$\mathrm{int}[\alpha \mapsto \tau] = \mathrm{int}$$
$$(\tau_1\ \tau_2)[\alpha \mapsto \tau] = (\tau_1[\alpha \mapsto \tau]\ \tau_2[\alpha \mapsto \tau])$$
$$\tau\!\downarrow_i^C\,[\alpha \mapsto \tau] = (\tau[\alpha \mapsto \tau])\!\downarrow_i^C$$

$$\mathtt{ff}[\alpha \mapsto \tau] = \mathtt{ff}$$
$$\mathtt{tt}[\alpha \mapsto \tau] = \mathtt{tt}$$
$$\phi_0 \wedge \phi_1[\alpha \mapsto \tau] = \phi_0[\alpha \mapsto \tau] \wedge \phi_1[\alpha \mapsto \tau]$$
$$\phi_0 \vee \phi_1[\alpha \mapsto \tau] = \phi_0[\alpha \mapsto \tau] \vee \phi_1[\alpha \mapsto \tau]$$
$$C \in \tau[\alpha \mapsto \tau] = C \in (\tau[\alpha \mapsto \tau])$$
$$C \notin \tau[\alpha \mapsto \tau] = C \notin (\tau[\alpha \mapsto \tau])$$
$$\forall \notin \tau[\alpha \mapsto \tau] = \forall \notin (\tau[\alpha \mapsto \tau])$$
$$(\tau_1\ @_\phi\ \tau_2)[\alpha \mapsto \tau] = (\tau_1[\alpha \mapsto \tau]\ @_\phi\ \tau_2[\alpha \mapsto \tau])$$

Fig. 10. Substitution of type variables.

Finally, we apply T-CONSTRUCTOR and conclude that

$$\Gamma, y : \tau_y \vdash C(\overline{e}) : C(\overline{\tau}) \ \& \ \bigvee \bar{\phi}$$

holds.

Case T-FUNAPP: We assume

$$\Gamma \vdash (e_1 \ e_2) : (\tau_1 \ @_\tau \tau_2) \ \& \ (\tau_1 \ @_\phi \tau_2) \vee \phi_1 \vee \phi_2 \vee \forall \notin \tau_1$$
$$y \notin \mathbf{dom}(\Gamma)$$

By inversion we get

$$\Gamma \vdash e_i : \tau_i \ \& \ \phi_i$$

for $i = 1, 2$ and apply the induction hypothesis on both of them. With

$$\Gamma, y : \tau_y \vdash e_i : \tau_i \ \& \ \phi_i$$

we can apply T-FUNAPP and conclude

$$\Gamma, y : \tau_y \vdash (e_1 \ e_2) : (\tau_1 \ @_\tau \tau_2) \ \& \ (\tau_1 \ @_\phi \tau_2) \vee \phi_1 \vee \phi_2 \vee \forall \notin \tau_1$$

Case T-REC: We assume

$$\Gamma \vdash \mathbf{rec} \ f_r \ x_r = e : \mathbf{rec} \ \alpha_r \ \tau_e = \phi_e \ \& \ \mathtt{ff}$$
$$y \notin \mathbf{dom}(\Gamma)$$

By inversion we get

$$\Gamma, x_r : \alpha_r, f_r : \mathbf{rec} \ \alpha_r \ \tau_e = \phi_e \vdash e : \tau_e \ \& \ \phi_e$$

By alpha-conversion we can assume that $x_r \neq y$ and $f_r \neq y$. Thus, with

$$y \notin \mathbf{dom}(\Gamma, x_r : \alpha_r, f_r : \mathbf{rec} \ \alpha_r \ \tau_e = \phi_e)$$

we apply the induction hypothesis and get

$$\Gamma, y : \tau_y, x_r : \alpha_r, f_r : \mathbf{rec} \ \alpha_r \ \tau_e = \phi_e \vdash e : \tau_e \ \& \ \phi_e$$

As all preconditions still hold, we can apply T-REC and finally conclude

$$\Gamma, y : \tau_y \vdash \mathbf{rec} \ f_r \ x_r = e : \mathbf{rec} \ \alpha_r \ \tau_e = \phi_e \ \& \ \mathtt{ff}$$

Case T-PATTERN-MATCHING: By assumption we have

$$\Gamma \vdash \mathbf{match} \ e_0 \ \mathbf{with} \ [C_i(\bar{x}) \to e_i] : \tau \ \& \ \phi_0 \vee \phi'$$
$$y \notin \mathbf{dom}(\Gamma)$$

We use inversion to deduce both

$$\mathtt{tt}; \tau_0; \Gamma \vdash_p [C_i(\bar{x}) \to e_i] : \tau \ \& \ \phi'$$

and
$$\Gamma \vdash e_0 : \tau_0 \;\&\; \phi_0$$

We can directly apply the induction hypothesis on both judgements and obtain
$$\mathtt{tt}; \tau_0; \Gamma, y : \tau_y \vdash_p [C_i(\bar{x}) \to e_i] : \tau \;\&\; \phi'$$

and
$$\Gamma, y : \tau_y \vdash e_0 : \tau_0 \;\&\; \phi_0$$

respectively. Applying T-Pattern-Matching concludes

$$\Gamma, y : \tau_y \vdash \mathbf{match}\; e_0 \;\mathbf{with}\; [C_i(\bar{x}) \to e_i] : \tau \;\&\; \phi_0 \vee \phi'$$

Case T-Pattern-Empty: Let $\Gamma' = \Gamma, y : \tau_y$ and

$$\phi_0; \tau_0; \Gamma' \vdash_p [\;] : \bot \;\&\; \phi_0$$

trivially holds by T-Pattern-Empty.

Case T-Pattern-Next: Assume

$$\phi_0; \tau_0; \Gamma \vdash_p [C(x_1, \ldots, x_n) \to e] \mathbin{+\!\!+} R : \tau' \cup \tau_e \;\&\; (\phi_0 \wedge C \in \tau_0 \wedge \phi_e) \vee \phi'$$
$$y \notin \mathbf{dom}(\Gamma)$$

By inversion we get

$$(\phi_0 \wedge C \notin \tau_0); \tau_0; \Gamma \vdash_p R : \tau' \;\&\; \phi'$$

and
$$\Gamma, x_i : \tau_0 {\downarrow}_i^C \vdash e : \tau_e \;\&\; \phi_e$$

for $i = 1, \ldots, n$. By alpha-conversion we know, that $\forall i = 1, \ldots, n.\; x_i \neq y$. Thus, we can apply the induction hypothesis on both judgements and get

$$(\phi_0 \wedge C \notin \tau_0; \tau_0; \Gamma, y : \tau_y \vdash_p R : \tau' \;\&\; \phi'$$

and
$$\Gamma, y : \tau_y, x_i : \tau_0 {\downarrow}_i^C \vdash e : \tau_e \;\&\; \phi_e$$

respectively. Now we apply T-Pattern-Next and conclude

$$\phi_0; \tau_0; \Gamma, y : \tau_y \vdash_p [C(x_1, \ldots, x_n) \to e] \mathbin{+\!\!+} R : \tau' \cup \tau_e \;\&\; (\phi_0 \wedge C \in \tau_0 \wedge \phi_e) \vee \phi'$$

Proof (Item 2). Proof by induction on the derivation of the analysis.

Case T-Integer: Let $\Gamma' = (\Gamma, y : \alpha)[\alpha \mapsto \tau_\alpha]$. By substitution and T-Integer our consequence
$$\Gamma' \vdash n : n[\alpha \mapsto \tau_\alpha] \;\&\; \mathtt{ff}[\alpha \mapsto \tau_\alpha]$$

immediately holds.

Case T-Identifier: We have two subcases.

Subcase $y = x$: We have to show that

$$(\Gamma, y : \alpha)[\alpha \mapsto \tau_\alpha] \vdash y : \tau_\alpha \ \& \ \text{ff}$$

holds. We apply inversion and get

$$((\Gamma, y : \alpha)[\alpha \mapsto \tau_\alpha])(y) = \tau_\alpha$$

By definition of substitution this is equivalent to

$$(\Gamma[\alpha \mapsto \tau_\alpha], y : \tau_\alpha)(y) = \tau_\alpha$$

As we require Γ to be well-formed this holds trivially.
Subcase $y \neq x$: As assumption we have

$$(\Gamma, y : \alpha)[\alpha \mapsto \tau_\alpha] \vdash x : \alpha' \ \& \ \text{ff}$$

By inversion we get $(\Gamma, y : \alpha)(x) = \alpha'$. Thus, we know that $\Gamma = \Gamma_1, x : \alpha', \Gamma_2$. As Γ is well-formed, we can insert a substitution without changing the equality

$$((\Gamma_1, x : \alpha', \Gamma_2, y : \alpha)[\alpha \mapsto \tau_\alpha])(x) = \alpha'$$

Now, we can apply T-IDENTIFIER and finally get

$$(\Gamma_1, x : \alpha', \Gamma_2, y : \alpha)[\alpha \mapsto \tau_\alpha] \vdash x : \alpha' \ \& \ \text{ff}$$

which is equivalent to our goal

$$(\Gamma, y : \alpha)[\alpha \mapsto \tau_\alpha] \vdash x : \alpha'[\alpha \mapsto \tau_\alpha] \ \& \ \text{ff}[\alpha \mapsto \tau_\alpha]$$

by the rules of substitution.
Case T-CONSTRUCTOR: On the assumption

$$\Gamma, y : \alpha \vdash C(\overline{e}) : C(\overline{\tau}) \ \& \ \vee \bar{\phi}$$

we apply inversion and get

$$\Gamma, y : \alpha \vdash e_i : \tau_i \ \& \ \phi_i$$

With the induction hypothesis, we can deduce

$$(\Gamma, y : \alpha)[\alpha \mapsto \tau_\alpha] \vdash e_i : \tau_i[\alpha \mapsto \tau_\alpha] \ \& \ \phi_i[\alpha \mapsto \tau_\alpha]$$

Applying T-CONSTRUCTOR yields

$$(\Gamma, y : \alpha)[\alpha \mapsto \tau_\alpha] \vdash C(\overline{e}) : C(\overline{\tau[\alpha \mapsto \tau_\alpha]}) \ \& \ \vee (\overline{\phi[\alpha \mapsto \tau_\alpha]})$$

which is, by rules of substitution, equivalent to

$$(\Gamma, y : \alpha)[\alpha \mapsto \tau_\alpha] \vdash C(\overline{e}) : C(\bar{\tau})[\alpha \mapsto \tau_\alpha] \ \& \ (\vee \bar{\phi})[\alpha \mapsto \tau_\alpha]$$

Case T-FUNAPP: On the assumption

$$\Gamma, y : \alpha \vdash (e_1 \ e_2) : (\tau_1 \ @_\tau \ \tau_2) \ \& \ (\tau_1 \ @_\phi \ \tau_2) \lor \phi_1 \lor \phi_2 \lor \forall \notin \tau_1$$

we apply inversion and get

$$\Gamma, y : \alpha \vdash e_i : \tau_i \ \& \ \phi_i$$

With the induction hypothesis, we can deduce

$$(\Gamma, y : \alpha)[\alpha \mapsto \tau_\alpha] \vdash e_i : \tau_i[\alpha \mapsto \tau_\alpha] \ \& \ \phi_i[\alpha \mapsto \tau_\alpha]$$

Applying T-FUNAPP yields

$$(\Gamma, y : \alpha)[\alpha \mapsto \tau_\alpha] \vdash (e_1 \ e_2) : (\tau_1[\alpha \mapsto \tau_\alpha] \ @_\tau \ \tau_2[\alpha \mapsto \tau_\alpha])$$
$$\& \ (\tau_1[\alpha \mapsto \tau_\alpha] \ @_\phi \ \tau_2[\alpha \mapsto \tau_\alpha]) \lor \phi_1[\alpha \mapsto \tau_\alpha]$$
$$\lor \phi_2[\alpha \mapsto \tau_\alpha] \lor \forall \notin \tau_1[\alpha \mapsto \tau_\alpha]$$

which is, by rules of substitution, equivalent to

$$(\Gamma, y : \alpha)[\alpha \mapsto \tau_\alpha] \vdash (e_1 \ e_2) : (\tau_1 \ @_\tau \ \tau_2)[\alpha \mapsto \tau_\alpha]$$
$$\& \ ((\tau_1 \ @_\phi \ \tau_2) \lor \phi_1 \lor \phi_2 \lor \forall \notin \tau_1)[\alpha \mapsto \tau_\alpha]$$

Case T-REC: We assume

$$\Gamma, y : \alpha \vdash \mathbf{rec} \ f_r \ x_r = e : \mu X. \forall \alpha_r \ [\phi_e] . \tau_e \ \& \ \mathbf{ff}$$

By inversion we can follow that $y \neq f_r$, $y \neq x_r$, and $\alpha \neq \alpha_r$. Additionally,

$$\Gamma, y : \alpha, f_r : \mu X. \forall \alpha_r \ [\phi_e] . \tau_e, x_r : \alpha_r \vdash e : \tau_e \ \& \ \phi_e$$

With the induction hypothesis we get

$$(\Gamma, y : \alpha, f_r : \mu X. \forall \alpha_r \ [\phi_e] . \tau_e, x_r : \alpha_r)[\alpha \mapsto \tau_\alpha] \vdash e : \tau_e[\alpha \mapsto \tau_\alpha] \ \& \ \phi_e[\tau \mapsto \tau_\alpha]$$

When we apply T-REC and exploit substitution we finally obtain

$$(\Gamma, y : \alpha)[\alpha \mapsto \tau_\alpha] \vdash \mathbf{rec} \ f_r \ x_r = e : (\mu X. \forall \alpha_r \ [\phi_e] . \tau_e)[\alpha \mapsto \tau_\alpha] \ \& \ \mathbf{ff}$$

Case T-PATTERN-MATCHING: We have the assumption

$$\Gamma, y : \alpha \vdash \mathbf{match} \ e_0 \ \mathbf{with} \ [C_i(\bar{x}) \to e_i] : \tau \ \& \ \phi_0 \lor \phi'$$

By inversion we get two judgements. On the first, $\Gamma, y : \alpha \vdash e_0 : \tau_0 \ \& \ \phi_0$, we apply the induction hypothesis and get

$$(\Gamma, y : \alpha)[\alpha \mapsto \tau_\alpha] \vdash e_0 : \tau_0[\alpha \mapsto \tau_\alpha] \ \& \ \phi_0[\alpha \mapsto \tau_\alpha]$$

On the second judgment

$$\mathbf{tt}; \tau_0; \Gamma, y : \alpha \vdash_p [C_i(\bar{x}) \to e_i] : \tau \ \& \ \phi'$$

we apply the induction hypothesis, too and obtain

$$\mathtt{tt}[\alpha \mapsto \tau_\alpha]; \tau_0[\alpha \mapsto \tau_\alpha]; (\Gamma, y : \alpha)[\alpha \mapsto \tau_\alpha] \vdash_p [C_i(\bar{x}) \to e_i]$$
$$: \tau[\alpha \mapsto \tau_\alpha] \mathbin{\&} \phi'[\alpha \mapsto \tau_\alpha]$$

Now we can apply T-PATTERN-MATCHING and conclude

$$(\Gamma, y : \alpha)[\alpha \mapsto \tau_\alpha] \vdash \mathbf{match}\ e_0\ \mathbf{with}\ [C_i(\bar{x}) \to e_i]$$
$$: \tau[\alpha \mapsto \tau_\alpha] \mathbin{\&} (\phi_0 \vee \phi')[\alpha \mapsto \tau_\alpha]$$

Case T-PATTERN-EMPTY: The consequent

$$\phi_0[\alpha \mapsto \tau_\alpha]; \tau_0[\alpha \mapsto \tau_\alpha]; (\Gamma, x : \alpha)[\alpha \mapsto \tau_\alpha] \vdash_p [\,]$$
$$: \perp[\alpha \mapsto \tau_\alpha] \mathbin{\&} \phi_0[\alpha \mapsto \tau_\alpha]$$

holds by T-PATTERN-EMPTY.

Case T-PATTERN-NEXT: As assumption we have

$$\phi_0; \tau_0; \Gamma, y : \alpha \vdash_p [C(x_1, \ldots, x_n) \to] +\!\!\!+ P$$
$$: \tau' \cup \tau_e \mathbin{\&} (\phi_0 \wedge C \in \tau_0 \wedge \phi_e) \vee \phi'$$

By inversion we get two judgments. As first we get

$$\Gamma, y : \alpha, x_i : \tau_0 \!\downarrow_i^C \vdash e : \tau_e \mathbin{\&} \phi_e$$

for $i = 1, \ldots, n$ with $\forall i : y \neq x_i$ by alpha conversion. Now we can apply the induction hypothesis and obtain

$$(\Gamma, y : \alpha, x_i : \tau_0 \!\downarrow_i^C)[\alpha \mapsto \tau_\alpha] \vdash e : \tau_e[\alpha \mapsto \tau_\alpha] \mathbin{\&} \phi_e[\alpha \mapsto \tau_\alpha]$$

for $i = 1, \ldots, n$. On the second judgment

$$(\phi_0 \wedge C \notin \tau_0); \tau_0; \Gamma, y : \alpha \vdash_p P : \tau' \mathbin{\&} \phi'$$

we apply the induction hypothesis and deduce

$$(\phi_0 \wedge C \notin \tau_0)[\alpha \mapsto \tau_\alpha]; \tau_0[\alpha \mapsto \tau_\alpha]; (\Gamma, y : \alpha)[\alpha \mapsto \tau_\alpha] \vdash_p P$$
$$: \tau'[\alpha \mapsto \tau_\alpha] \mathbin{\&} \phi'[\alpha \mapsto \tau_\alpha]$$

Finally, we apply T-PATTERN-NEXT and conclude that

$$\phi_0[\alpha \mapsto \tau_\alpha]; \tau_0[\alpha \mapsto \tau_\alpha]; \Gamma, y : \alpha \vdash_p [C(x_1, \ldots, x_n) \to e] +\!\!\!+ P$$
$$: (\tau' \cup \tau_e)[\alpha \mapsto \tau_\alpha] \mathbin{\&} ((\phi_0 \wedge \tau_0 \in \wedge \phi_e) \vee \phi')[\alpha \mapsto \tau_\alpha]$$

holds.

Proof (Item 2). Proof by induction on the derivation of the analysis.

Case T-INTEGER: The consequent $\Gamma \vdash n[y \mapsto v] : n$ & ff trivially holds by T-INTEGER.

Case T-IDENTIFIER: For this case we have two subcases. One with $y = x$ and one with $y \neq x$.

Subcase $y = x$: We have the assumptions

$$\Gamma, y : \tau_y \vdash y : \tau \ \& \ \text{ff}$$
$$\Gamma \vdash v : \tau_y \ \& \ \text{ff}$$

By inversion we get $(\Gamma, y : \tau_y)(y) = \tau$ and thus, $\tau = \tau_y$ and $\Gamma \vdash v : \tau \ \& \ \text{ff}$. By applying substitution on our goal

$$\Gamma \vdash y[y \mapsto v] : \tau \ \& \ \text{ff}$$

we get $\Gamma \vdash v : \tau \ \& \ \text{ff}$ and are done.

Subcase $y \neq x$: We have the assumptions

$$\Gamma, y : \tau_y \vdash x : t \ \& \ \text{ff}$$
$$\Gamma \vdash v : \tau_y \ \& \ \text{ff}$$

By inversion we get $(\Gamma, y : \tau_y)(x) = t$. Thus, $\Gamma = \Gamma_1, x : t, \Gamma_2$ which leads to $(\Gamma_1, x : t, \Gamma_2)(x) = t$. This is a precondition for $\Gamma, x : t \vdash x : t \ \& \ \text{ff}$ which by rules of substitution is equivalent to $\Gamma, x : t \vdash x[y \mapsto v] : t \ \& \ \text{ff}$.

Case T-CONSTRUCTOR: Assume

$$\Gamma, y : \tau_y \vdash C(\overline{e}) : C(\overline{\tau}) \ \& \ \vee \overline{\phi}$$

By inversion we get

$$\Gamma, y : \tau_y \vdash e_i : \tau_i \ \& \ \phi_i$$

for $i = 1, \ldots, n$ and apply the induction hypothesis. Thus, we obtain

$$\Gamma \vdash e_i[y \mapsto v] : \tau_i \ \& \ \phi_i$$

and can apply T-CONSTRUCTOR and conclude

$$\Gamma \vdash C(\overline{e[y \mapsto v]}) : C(\overline{\tau}) \ \& \ \vee \overline{\phi}$$

which is by substitution equivalent to

$$\Gamma \vdash C(\overline{e})[y \mapsto v] : C(\overline{\tau}) \ \& \ \vee \overline{\phi}$$

Case T-FUNAPP: Assume

$$\Gamma, y : \tau_y \vdash (e_1 \ e_2) : (\tau_1 \ @_\tau \ \tau_2) \ \& \ (\tau_1 \ @_\phi \ \tau_2) \vee \phi_1 \vee \phi_2 \vee \forall \notin \tau_1$$

By inversion we get

$$\Gamma, y : \tau_y \vdash e_i : \tau_i \ \& \ \phi_i$$

and apply the induction hypothesis. Thus, we obtain

$$\Gamma \vdash e_i[y \mapsto v] : \tau_i \;\&\; \phi_i$$

and can apply T-FUNAPP and conclude

$$\Gamma \vdash (e_1[y \mapsto v] \; e_2[y \mapsto v]) : (\tau_1 \;@_\tau\; \tau_2) \;\&\; (\tau_1 \;@_\phi\; \tau_2) \vee \phi_1 \vee \phi_2 \vee \forall \notin \tau_1$$

which is equivalent, by the rules of substitution, to

$$\Gamma \vdash ((e_1 \; e_2))[y \mapsto v] : (\tau_1 \;@_\tau\; \tau_2) \;\&\; (\tau_1 \;@_\phi\; \tau_2) \vee \phi_1 \vee \phi_2 \vee \forall \notin \tau_1$$

Case T-REC: As part of the antecedent we have $\Gamma, y : \tau_y \vdash \mathbf{rec}\; f_r\; x_r = e : \mathbf{rec}\; \alpha_r\; \tau_e = \phi_e \;\&\; \mathtt{ff}$. By inversion we get $\Gamma, y : \tau_y, f_r : \mathbf{rec}\; \alpha_r\; \tau_e = \phi_e, x_r : \alpha_r \vdash e : \tau_e \;\&\; \phi_e$. Our induction hypothesis contains the antecedent $\Gamma \vdash v : \tau_y \;\&\; \mathtt{ff}$ but we need $\Gamma, f_r : \mathbf{rec}\; \alpha_r\; \tau_e = \phi_e \vdash v : \tau_y \;\&\; \mathtt{ff}$ which we get using weakening (Item 2). Now we can apply the induction hypothesis and obtain $\Gamma, f_r : \mathbf{rec}\; \alpha_r\; \tau_e = \phi_e, x_r : \alpha_r \vdash e[y \mapsto v] : \tau_e \;\&\; \phi_e$. Applying T-REC and the rules of substitution finally yields $\Gamma \vdash (\mathbf{rec}\; f_r\; x_r = e)[y \mapsto v] : \mathbf{rec}\; \alpha_r\; \tau_e = \phi_e \;\&\; \mathtt{ff}$.

Case T-PATTERN-MATCHING: We have $\Gamma, y : \tau_y \vdash \mathbf{match}\; e_0\; \mathbf{with}\; [C(\overline{x}) \to e_i] : t \;\&\; \phi_0 \vee \phi'$. By inversion we obtain $\mathtt{tt}; \tau_0; \Gamma, y : \tau_y \vdash_p [C(\overline{x}) \to e_i] : t \;\&\; \phi'$ and $\Gamma, y : \tau_y \vdash e_0 : \tau_0 \;\&\; \phi_0$. We apply the induction hypothesis on both preconditions and get $\mathtt{tt}; \tau_0; \Gamma \vdash_p \{C_i\bar{x} \to e_i\}[y \mapsto v] : t \;\&\; \phi'$ and $\Gamma \vdash e_0[y \mapsto v] : \tau_0 \;\&\; \phi_0$ which are the preconditions for T-PATTERN-MATCHING. We conclude by applying the rules of substitution and get $\Gamma \vdash (\mathbf{match}\; e_0\; \mathbf{with}\; [C(\overline{x}) \to e_i])[y \mapsto v] : t \;\&\; \phi_0 \vee \phi'$.

Case T-PATTERN-EMPTY: Our goal, $\phi_0; \tau_0; \Gamma \vdash_p \{\}[y \mapsto v] : \bot \;\&\; \phi_0$, trivially holds by substitution and T-PATTERN-EMPTY.

Case T-PATTERN-NEXT: We assume $\phi_0; \tau_0; \Gamma, y : \tau_y \vdash_p \{C(\bar{x}) \to e\}; \bar{p} : t' \cup \tau_e \;\&\; (\phi_0 \wedge C \in \tau_0 \wedge \phi_e) \vee \phi'$. By inversion we get (1) $(\phi_0 \wedge C \notin \tau_0); \tau_0; \Gamma, y : \tau_y \vdash_p \bar{p} : t' \;\&\; \phi'$ and (2) $\Gamma, y : \tau_y, x_i : \tau_0 \downarrow_i^C \vdash e : \tau_e \;\&\; \phi_e$ with $i = 1, \ldots, n$. On (1) we can apply the induction hypothesis and get $(\phi_0 \wedge C.n \nsubseteq \tau_0); \tau_0; \Gamma \vdash_p \bar{p}[y \mapsto v] : t' \;\&\; \phi'$. On (2) the antecedent of the induction hypothesis does not fit $\Gamma \vdash v : \tau_y \;\&\; \mathtt{ff}$. Thus we apply weakening (Item 2) and obtain $\Gamma, x_i : \tau_0 \downarrow_i^C \vdash v : \tau_y \;\&\; \mathtt{ff}$ with $i = 1, \ldots, n$. Now we can apply the induction hypothesis and get $\Gamma, x_i : \tau_0 \downarrow_i^C \vdash e[y \mapsto v] : \tau_e \;\&\; \phi_e$. Using the rule T-PATTERN-NEXT and substitution, we conclude that $\phi_0; \tau_0; \Gamma \vdash_p (\{C(\bar{x}) \to e\}; \bar{p})[y \mapsto v] : t' \cup \tau_e \;\&\; (\phi_0 \wedge C \in \tau_0 \wedge \phi_e) \vee \phi'$ holds.

Proof (Lemma 5). Proof by induction on the derivation of $\vdash_p$.

Case T-PATTERN-EMPTY: For

$$\mathtt{ff}; \tau_0; \Gamma \vdash_p [\;] : \bot \;\&\; \mathtt{ff}$$

we have $\nvDash \mathtt{ff}$.

Case T-PATTERN-NEXT: We have

$$\phi_0; \tau_0; \Gamma \vdash_p [C(x_1, \ldots, x_n) \rightarrow e] + P : \tau' \cup \tau_e \ \& \ (\phi_0 \wedge C \in \tau_0 \wedge \phi_e) \vee \phi'$$

and know that $\phi_0 = \mathtt{ff}$. Thus, the crash condition simplifies to $(\mathtt{ff} \wedge \tau_0 \in \wedge \phi_e) \vee \phi' = \phi'$. We have to show that $\nvDash \phi'$. By inversion we get

$$(\phi_0 \wedge C \notin \tau_0); \tau_0; \Gamma \vdash_p P : \tau' \ \& \ \phi'$$

and with $\phi_0 = \mathtt{ff}$ we get

$$\mathtt{ff}; \tau_0; \Gamma \vdash_p P : \tau' \ \& \ \phi'$$

Applying the induction hypothesis finally yields $\nvDash \phi'$.

Proof (Theorem 1). We abbreviate $\mathcal{E}[\hat{e}]$ with e and $\mathcal{E}[\hat{e}']$ with e', respectively. By induction on the derivation of $e \hookrightarrow e'$. In all the cases building on EVAL-HOLE, we have $\hat{e} \hookrightarrow \hat{e}'$ by inversion on $e \hookrightarrow e'$.

Case $e = C(v_1 \ldots, v_n, \hat{e}, e_1, \ldots, e_m) \hookrightarrow e' = C(v_1 \ldots, v_n, \hat{e}', e_1, \ldots, e_m)$: By assumption we have

$$\Gamma \vdash C(v_{1\ldots n}, \hat{e}, e_{1\ldots m}) : C(\tau_{v_{1\ldots n}}, \hat{\tau}, \tau_{e_{1\ldots m}}) \ \& \ \vee_{i=1}^n \phi_{v_i} \vee \hat{\phi} \vee \vee_{i=1}^m \phi_{e_i} \quad (1)$$

$$\Gamma \vdash C(v_{1\ldots n}, \hat{e}', e_{1\ldots m}) : C(\tau'_{v_{1\ldots n}}, \hat{\tau}', \tau'_{e_{1\ldots m}}) \ \& \ \vee_{i=1}^n \phi'_{v_i} \vee \hat{\phi}' \vee \vee_{i=1}^m \phi'_{e_i} \quad (2)$$

Inversion on equation (1) yields

$$\Gamma \vdash v_i : \tau_{v_i} \ \& \ \phi_{v_i} \qquad \Gamma \vdash e_i : \tau_{e_i} \ \& \ \phi_{e_i} \qquad \Gamma \vdash \hat{e} : \hat{\tau} \ \& \ \hat{\phi}$$

and for inversion on equation (2) we get

$$\Gamma \vdash v_i : \tau'_{v_i} \ \& \ \phi'_{v_i} \qquad \Gamma \vdash e_i : \tau'_{e_i} \ \& \ \phi'_{e_i} \qquad \Gamma \vdash \hat{e}' : \hat{\tau}' \ \& \ \hat{\phi}'$$

Thus, we can follow that $\tau_{v_i} = \tau'_{v_i}$, $\tau_{e_i} = \tau'_{e_i}$, $\phi_{v_i} = \phi'_{v_i}$, and $\phi_{e_i} = \phi'_{e_i}$. Additionally, we have $(\hat{\tau} = \hat{\tau}' \wedge \hat{\phi} \leftrightarrow \hat{\phi}') \rightarrow t = t' \wedge \phi = \phi'$. Now we can apply the induction hypothesis $(\Gamma \vdash \hat{e} : \hat{\tau} \ \& \ \hat{\phi} \wedge \hat{e} \hookrightarrow \hat{e}' \wedge \Gamma \vdash \hat{e}' : \hat{\tau}' \ \& \ \hat{\phi}' \rightarrow \hat{\tau} = \hat{\tau}' \wedge \hat{\phi} \leftrightarrow \hat{\phi}')$ and thus conclude that $t = t'$ and $\phi \leftrightarrow \phi'$.

Case $e = (\hat{e} \ e_2) \hookrightarrow e' = (\hat{e}' \ e_2)$: By assumption we have

$$\Gamma \vdash (\hat{e} \ e_2) : (\hat{\tau} @_\tau \tau_2) \ \& \ (\hat{\tau} @_\phi \tau_2) \vee \hat{\phi} \vee \phi_2 \vee \forall \notin \hat{\tau} \quad (3)$$

$$\Gamma \vdash (\hat{e}' \ e_2) : (\hat{\tau}' @_\tau \tau'_2) \ \& \ (\hat{\tau}' @_\phi \tau'_2) \vee \hat{\phi}' \vee \phi'_2 \vee \forall \notin \hat{\tau}' \quad (4)$$

When applying inversion on equation (3) we obtain

$$\Gamma \vdash \hat{e} : \hat{\tau} \ \& \ \hat{\phi} \qquad \Gamma \vdash e_2 : \tau_2 \ \& \ \phi_2$$

and for inversion on equation (4) we get

$$\Gamma \vdash \hat{e}' : \hat{\tau}' \ \& \ \hat{\phi}' \qquad \Gamma \vdash e_2 : \tau'_2 \ \& \ \phi'_2$$

Thus, we have $\tau_2 = \tau'_2$ and $\phi_2 = \phi'_2$ and $(\hat{\tau} = \hat{\tau}' \wedge \hat{\phi} \leftrightarrow \hat{\phi}') \rightarrow t = t' \wedge \phi = \phi'$. We apply the induction hypothesis $(\Gamma \vdash \hat{e} : \hat{\tau} \ \& \ \hat{\phi} \wedge \hat{e} \hookrightarrow \hat{e}' \wedge \Gamma \vdash \hat{e}' : \hat{\tau}' \ \& \ \hat{\phi}' \rightarrow \hat{\tau} = \hat{\tau}' \wedge \hat{\phi} \leftrightarrow \hat{\phi}')$ and conclude that $t = t'$ and $\phi \leftrightarrow \phi'$.

Case $e = (v\ \hat{e}) \hookrightarrow e' = (v\ \hat{e}')$: By assumption we have

$$\Gamma \vdash (v\ \hat{e})\ : (\tau_1\ @_\tau\ \hat{\tau})\ \&\ (\tau_1\ @_\phi\ \hat{\tau}) \vee \phi_1 \vee \hat{\phi} \vee \forall \notin \tau_1 \tag{5}$$

$$\Gamma \vdash (v\ \hat{e}') : (\tau_1\ @_\tau\ \hat{\tau}')\ \&\ (\tau_1\ @_\phi\ \hat{\tau}') \vee \phi_1 \vee \hat{\phi}' \vee \forall \notin \tau_1 \tag{6}$$

Inversion on equation (5) yields

$$\Gamma \vdash \hat{e} : \hat{\tau}\ \&\ \hat{\phi} \qquad \Gamma \vdash v : \tau_1\ \&\ \phi_1$$

and inversion on equation (6) yields

$$\Gamma \vdash \hat{e}' : \hat{\tau}'\ \&\ \hat{\phi}' \qquad \Gamma \vdash v : \tau_1'\ \&\ \phi_1'$$

Thus we conclude $\tau_1 = \tau_1'$, $\phi_1 = \phi_1'$, and furthermore, $(\hat{\tau} = \hat{\tau}' \wedge \hat{\phi} \leftrightarrow \hat{\phi}') \rightarrow t = t' \wedge \phi = \phi'$. We apply the induction hypothesis ($\Gamma \vdash \hat{e} : \hat{\tau}\ \&\ \hat{\phi} \wedge \hat{e} \hookrightarrow \hat{e}' \wedge \Gamma \vdash \hat{e}' : \hat{\tau}'\ \&\ \hat{\phi}' \rightarrow \hat{\tau} = \hat{\tau}' \wedge \hat{\phi} \leftrightarrow \hat{\phi}'$) and conclude that $t = t'$ and $\phi \leftrightarrow \phi'$.

Case $e = \textbf{match}\ \hat{e}\ \textbf{with}\ [C(\overline{x}) \rightarrow e] \hookrightarrow e' = \textbf{match}\ \hat{e}'\ \textbf{with}\ [C(\overline{x}) \rightarrow e]$: By assumption we have

$$\Gamma \vdash \textbf{match}\ \hat{e}\ \textbf{with}\ [C(\overline{x}) \rightarrow e]\ : \tau_p\ \&\ \hat{\phi} \vee \phi_p \tag{7}$$

$$\Gamma \vdash \textbf{match}\ \hat{e}'\ \textbf{with}\ [C(\overline{x}) \rightarrow e] : \tau_p'\ \&\ \hat{\phi}' \vee \phi_p' \tag{8}$$

By inversion on equation (7) we get

$$\Gamma \vdash \hat{e} : \hat{\tau}\ \&\ \hat{\phi}$$
$$\texttt{tt}; \hat{\tau}; \Gamma \vdash_p [C(\overline{x}) \rightarrow e] : \tau_p\ \&\ \phi_p \tag{9}$$

and by inversion on equation (8) we obtain

$$\Gamma \vdash \hat{e}' : \hat{\tau}'\ \&\ \hat{\phi}'$$
$$\texttt{tt}; \hat{\tau}'; \Gamma \vdash_p [C(\overline{x}) \rightarrow e] : \tau_p'\ \&\ \phi_p' \tag{10}$$

Now by applying the induction hypothesis ($\Gamma \vdash \hat{e} : \hat{\tau}\ \&\ \hat{\phi} \wedge \hat{e} \hookrightarrow \hat{e}' \wedge \Gamma \vdash \hat{e}' : \hat{\tau}'\ \&\ \hat{\phi}' \rightarrow \hat{\tau} = \hat{\tau}' \wedge \hat{\phi} \leftrightarrow \hat{\phi}'$) we can conclude that the equations (9) and (10) are equivalent and thus, $t = t'$ and $\phi = \phi'$.

Now to the interesting cases.

Case $e = ((\textbf{rec}\ f\ x = \hat{e})\ v) \hookrightarrow e' = \hat{e}[x \mapsto v, f \mapsto \textbf{rec}\ f\ x = \hat{e}]$: By assumption we have

$$\Gamma \vdash ((\textbf{rec}\ f\ x = \hat{e}\ v)) : (\tau_1\ @_\tau\ \tau_2)\ \&\ (\tau_1\ @_\phi\ \tau_2) \vee \phi_1 \vee \phi_2 \vee \forall \notin \tau_1 \tag{11}$$

on which we apply inversion (T-FunApp) and get

$$\Gamma \vdash v : \tau_v\ \&\ \texttt{ff}$$
$$\Gamma \vdash \textbf{rec}\ f\ x = \hat{e} : \textbf{rec}\ \alpha\ \tau_e = \phi_e\ \&\ \texttt{ff} \tag{12}$$

Applying inversion (T-REC) on the second judgement (12) yields

$$\Gamma, x : \alpha, f : \mathbf{rec}\ \alpha\ \tau_e = \phi_e \vdash \hat{e} : \tau_e\ \&\ \phi_e \qquad \alpha\ \text{fresh} \qquad (13)$$

Thus, for equation (11) we have $(\tau_1 @_\tau \tau_2) = ((\mathbf{rec}\ \alpha\ \tau_e = \phi_e)@_\tau \tau_v) = \tau_e[\alpha \mapsto \tau_v]$ and $(\tau_1 @_\phi \tau_2) \vee \phi_1 \vee \phi_2 \vee \forall \notin \tau_1 = ((\mathbf{rec}\ \alpha\ \tau_e = \phi_e)@_\phi \tau_v) \vee \mathtt{ff} \vee \mathtt{ff} \vee \forall \notin (\mathbf{rec}\ \alpha\ \tau_e = \phi_e) = \phi_e[\alpha \mapsto \tau_v]$. Our assumption can thus be rewritten as

$$\Gamma \vdash ((\mathbf{rec}\ f\ x = \hat{e}\ v)) : \tau_e[\alpha \mapsto \tau_v]\ \&\ \phi_e[\alpha \mapsto \tau_v]$$

We now claim that the following holds

$$\Gamma \vdash \hat{e}[x \mapsto v, f \mapsto \mathbf{rec}\ f\ x = \hat{e}] : \tau_e[\alpha \mapsto \tau_v]\ \&\ \phi_e[\alpha \mapsto \tau_v]$$

To show that, we apply Item 2 and get

$$\Gamma, x : \alpha, f : \mathbf{rec}\ \alpha\ \tau_e = \phi_e \vdash \hat{e} : \tau_e\ \&\ \phi_e\ \wedge\ \Gamma \vdash \mathbf{rec}\ f\ x = \hat{e} : \mathbf{rec}\ \alpha\ \tau_e = \phi_e\ \&\ \mathtt{ff} \rightarrow$$
$$\Gamma, x : \alpha \vdash \hat{e}[f \mapsto \mathbf{rec}\ f\ x = \hat{e}] : \tau_e\ \&\ \phi_e$$

On the consequent, we apply Item 2 and get

$$\Gamma, x : \alpha \vdash \hat{e}[f \mapsto \mathbf{rec}\ f\ x = \hat{e}] : \tau_e\ \&\ \phi_e \rightarrow$$
$$\Gamma[\alpha \mapsto \tau_v], x : \tau_v \vdash \hat{e}[f \mapsto \mathbf{rec}\ f\ x = \hat{e}] : \tau_e[x\alpha \mapsto \tau_v]\ \&\ \phi_e[\alpha \mapsto \tau_v]$$

From equation (13) we know that α was fresh, and thus $\alpha \notin ran(\Gamma)$ which implies $\Gamma[\alpha \mapsto \tau_v] = \Gamma$. Finally, we again apply Item 2 and get

$$\Gamma[\alpha \mapsto \tau_v], x : \tau_v \vdash \hat{e}[f \mapsto \mathbf{rec}\ f\ x = \hat{e}] : \tau_e[x\alpha \mapsto \tau_v]\ \&\ \phi_e[\alpha \mapsto \tau_v]$$
$$\wedge\ \Gamma \vdash v : \tau_v\ \&\ \mathtt{ff} \rightarrow$$
$$\Gamma \vdash (\hat{e}[f \mapsto \mathbf{rec}\ f\ x = \hat{e}])[x \mapsto v] : \tau_e[\alpha \mapsto \tau_v]\ \&\ \phi_e[\alpha \mapsto \tau_v]$$

As $x \notin free(\mathbf{rec}\ f\ x = \hat{e})$, we can reorder the substitution and

$$\Gamma \vdash \hat{e}[x \mapsto v, f \mapsto \mathbf{rec}\ f\ x = \hat{e}] : \tau_e[\alpha \mapsto \tau_v]\ \&\ \phi_e[\alpha \mapsto \tau_v]$$

holds, as claimed.

Case $e = \mathbf{match}\ C(\overline{v})\ \mathbf{with}\ C(\overline{x}) \rightarrow \hat{e}|r \hookrightarrow e' = \hat{e}[x_i \mapsto v_i]\ (i = 1 \ldots, n)$: By our assumptions we know that

$$\Gamma \vdash \mathbf{match}\ C(\overline{v})\ \mathbf{with}\ C(\overline{x}) \rightarrow \hat{e}|r : \tau_p\ \&\ \phi_0 \vee \phi_p \qquad (14)$$
$$\Gamma \vdash \hat{e}[x_i \mapsto v_i] : t'\ \&\ \phi' \qquad (15)$$

Now we have to show that $\tau_p = t'$ and $\phi_0 \vee \phi_p \leftrightarrow \phi'$ holds. By inversion on equation (14) we get two judgements: At first, $\Gamma \vdash C(\overline{v}) : C(\overline{\tau_v})\ \&\ \mathtt{ff}$ which by inversion yields

$$\Gamma \vdash v_i : \tau_i\ \&\ \mathtt{ff} \qquad i = 1, \ldots, n \qquad (16)$$

and second, we get

$$\texttt{tt}; C(\overline{\tau_v}); \Gamma \vdash_p C(\bar{x}) \to \hat{e}|r : \tau_e \cup \tilde{\tau} \ \& \ (\phi_0 \wedge C \notin C(\overline{\tau_v})) \wedge \phi_e) \vee \tilde{\phi} \quad (17)$$

We again apply inversion and by T-PATTERN-NEXT get

$$(\phi_0 \wedge C \notin C(\overline{\tau_v})); C(\overline{\tau_v}); \Gamma \vdash_p r : \tilde{\tau} \ \& \ \tilde{\phi} \quad (18)$$

$$\Gamma, x_i : C(\overline{\tau_v}) \downarrow_i^C \vdash \hat{e} : \tau_e \ \& \ \phi_e \qquad i = 1, \dots, n \quad (19)$$

As we know that $\phi_0 = \texttt{tt}$, $C \in C(\overline{\tau_v}) = \texttt{tt}$, and $C \notin C(\overline{\tau_v}) = \texttt{ff}$ (the pattern matches), we can change equation (18) to

$$\texttt{ff}; C(\overline{\tau_v}); \Gamma \vdash_p r : \tilde{\tau} \ \& \ \tilde{\phi}$$

and then apply Lemma 5 and conclude $\nvDash \tilde{\phi}$. Thus, equation (17) simplifies to

$$\texttt{tt}; C(\overline{\tau_v}); \Gamma \vdash_p C(\bar{x}) \to \hat{e}|r : \tau_e \cup \tilde{\tau} \ \& \ \phi_e$$

Now we have to derive the types of equation (15) and show that $\tau_e \cup \tilde{\tau} = t'$ and $\phi_e \leftrightarrow \phi'$. In equation (19) we can simplify $C(\overline{\tau_v}) \downarrow_i^C$ to τ_i and then apply Item 2 (value substitution) n times using equation (16) with the appropriate i. Thus, we obtain

$$\Gamma \vdash \hat{e}[x_n \mapsto v_n] \dots [x_1 \mapsto v_1] : \tau_e \ \& \ \phi_e$$

The substitution is order-independent as $\forall i, j : x_i \notin v_i$ and thus we get

$$\Gamma \vdash \hat{e}[x_i \mapsto v_i] : \tau_e \ \& \ \phi_e$$

By subtyping ($\tau_e \leq \tau_e \cup \tilde{\tau}$) our claim holds.

Case $e = \textbf{match } C(\overline{v}) \textbf{ with } D(\bar{x}) \to \hat{e}|r \hookrightarrow e' = \textbf{match } C(\overline{v}) \textbf{ with } r$: By assumption we have

$$\Gamma \vdash \textbf{match } C(\overline{v}) \textbf{ with } D(\bar{x}) \to \hat{e}|r : \tau_p \ \& \ \phi_0 \vee \phi_p \quad (20)$$

$$\Gamma \vdash \textbf{match } C(\overline{v}) \textbf{ with } r : \tau_p' \ \& \ \phi_0' \vee \phi_p' \quad (21)$$

Our claim is that $\tau_p' \leq \tau_p$ and $\phi_0 \vee \phi_p \leftrightarrow \phi_0' \vee \phi_p'$. To prove this, we apply inversion on equation (20) and obtain

$$\Gamma \vdash C(\overline{v}) : C(\overline{\tau}) \ \& \ \texttt{ff} \quad (22)$$

$$\texttt{tt}; C(\overline{\tau}); \Gamma \vdash_p D(\bar{x}) \to \hat{e}|r : \tau_{\hat{e}} \cup \tilde{\tau} \ \& \ \texttt{tt} \wedge C \in D \wedge \phi_{\hat{e}} \vee \tilde{\phi} \quad (23)$$

where the crash condition ($\phi_0 = \texttt{ff}$) of equation (22) can be deduced by another inversion. Equation (23) can be simplified as we know that $C \in D = \texttt{ff}$. Thus, we get $\phi = \phi_p$ and (without simplifications) $t = \tau_p$. Inversion on the other assumption (21) again yields equation (21) and additionally we have

$$\texttt{tt}; C(\overline{\tau}); \Gamma \vdash_p r : \tau_p' \ \& \ \phi_p' \quad (24)$$

With the same reason as above, we can follow that $\phi_0 = \phi_0' = \texttt{ff}$ and thus $\phi' = \phi_p'$. Inversion on both equations (23) and (24) requires a case distinction over r as both the rules T-PATTERN-EMPTY and T-PATTERN-NEXT match.

Subcase $r = [\ ]$: Inversion on equation (23) yields

$$\texttt{tt} \wedge (D \in C(\overline{v}) \vee D \notin C(\overline{\tau})); C(\overline{\tau}); \Gamma \vdash_p [\ ] : \tilde{\tau} \ \& \ \tilde{\phi}$$

By T-PATTERN-EMPTY we get $\tilde{\tau} = \bot$ and $\tilde{\phi} = \texttt{tt}$. Thus, $t = \tau_{\hat{e}} \cup \hat{\tau}$ and $\phi = \tilde{\phi} = \texttt{tt}$. With $r = [\ ]$ and T-PATTERN-EMPTY equation (24) gets more specific: $\texttt{tt}; C(\overline{\tau}); \Gamma \vdash_p [\ ] : \bot \ \& \ \texttt{tt}$. It follows that $\tau'_p = \bot$ and $\phi' = \texttt{tt}$ and thus, our claim holds.

Subcase $r \neq [\ ]$: Inversion on equation (23) now gives

$$\texttt{tt} \wedge (D \in C(\overline{v}) \vee D \notin C(\overline{\tau})); C(\overline{\tau}); \Gamma \vdash_p r : \tilde{\tau} \ \& \ \tilde{\phi}$$

This can be simplified by $D \notin C(\overline{\tau}) = \texttt{tt}$ to

$$\texttt{tt}; C(\overline{\tau}); \Gamma \vdash_p r : \tilde{\tau} \ \& \ \tilde{\phi} \tag{25}$$

As we have deterministic rules, we can follow from the equivalency of equation (24) and (25) that $\tilde{\tau} = \tau'_p$ and $\tilde{\phi} = \phi'_p$. Thus, $\phi = \phi'$ and $t = t'$.

And finally, the error cases.

Case $e = (\hat{v} \ v) \hookrightarrow e' = \text{err}$: By assumption we have

$$\Gamma \vdash (\hat{v} \ v) : t \ \& \ \phi \tag{26}$$
$$\Gamma \vdash \text{err} : \bot \ \& \ \texttt{tt}$$

By inversion on equation (26) we also get $\Gamma \vdash \hat{v} : \tau_{\hat{v}} \ \& \ \texttt{ff}$. As $\hat{v} \in \{n, C(\overline{v})\}$, we know that $\forall \notin \tau_{\hat{v}}$ holds and thus, $\phi = \texttt{tt}$. But then, $\phi = \phi'$ and $t' \leq t$.

Case $e = ((\textbf{rec} \ f \ x = e) \ \text{err}) \hookrightarrow e' = \text{err}$: By assumption we have

$$\Gamma \vdash ((\textbf{rec} \ f \ x = e) \ \text{err}) : t \ \& \ \phi \tag{27}$$
$$\Gamma \vdash \text{err} : \bot \ \& \ \texttt{tt}$$

By inversion on equation (27) we also get $\Gamma \vdash \text{err} : \bot \ \& \ \texttt{tt}$. Thus, $\phi_2 = \texttt{tt}$ which implies $\phi = \texttt{tt}$. But then, $\phi = \phi'$ and $t' \leq (\tau_1 \ @_\tau \ \tau_2)$.

Case $e = \textbf{match} \ \text{err} \ \textbf{with} \ P \hookrightarrow e' = \text{err}$: by assumption we have

$$\Gamma \vdash \textbf{match} \ \text{err} \ \textbf{with} \ P : t \ \& \ \phi \tag{28}$$
$$\Gamma \vdash \text{err} : \bot \ \& \ \texttt{tt}$$

by inversion on equation (28) we also get $\Gamma \vdash \text{err} : \bot \ \& \ \texttt{tt}$. Thus, $\phi = \texttt{tt}$. But then, $\phi = \phi'$ and $t' \leq \tau_p$.

Case $e = \textbf{match} \ C(\overline{v}) \ \textbf{with} \ [\ ] \hookrightarrow e' = \text{err}$: by assumption we have

$$\Gamma \vdash \textbf{match} \ C(\overline{v}) \ \textbf{with} \ [\ ] : t \ \& \ \phi \tag{29}$$
$$\Gamma \vdash \text{err} : \bot \ \& \ \texttt{tt}$$

by inversion on equation (29) we also get $\texttt{tt}; \tau_0; \Gamma \vdash_p [\ ] : \tau_p \ \& \ \phi_p$. By T-PATTERN-EMPTY we know that $\tau_p = \bot$ and $\phi = \texttt{tt}$. Thus, $t = \bot$ and $\phi = \texttt{tt}$. But then, $\phi = \phi'$ and $t' \leq t$.

Case $e = C(v_1, \ldots, v_n, \mathrm{err}, e_1, \ldots, e_m) \hookrightarrow e' = \mathrm{err}$: Assumptions give us

$$\Gamma \vdash C(v_1, \ldots, v_n, \mathrm{err}, e_1, \ldots, e_m) : t \;\&\; \phi \qquad (30)$$
$$\Gamma \vdash \mathrm{err} : \bot \;\&\; \mathtt{tt}$$

By inversion on equation (30) we get, amongst others, $\Gamma \vdash \mathrm{err} : \bot \;\&\; \mathtt{tt}$, and thus, we can follow by T-CONSTRUCTOR, that $\phi = \mathtt{tt}$. We can now conclude $t' \leq t$ and $\phi = \phi'$.

Proof (Item 2). For the simple cases $e = \{n, \mathbf{rec}\ f\ x = e, C(\overline{v}), x\}$ the assumption does not hold, as $\Gamma \vdash e : \tau \;\&\; \mathtt{ff}$.

Case $e = \mathrm{err}$: Trivially holds.

Case $e = C(\overline{e})$: Holds by inversion, application of the induction hypothesis and application of SCTORERR.

Case $e = (e_1\ e_2)$:
  - Wlog assume $\vDash \mathcal{T}(\phi_1)$: Holds by inversion, application of the induction hypothesis and application of SAPPERR1
  - Wlog assume $\vDash \mathcal{T}(\phi_2)$: Holds by inversion, application of the induction hypothesis and application of SAPPERR2.
  - Wlog assume $\vDash \mathcal{T}(\forall \notin \tau_1)$: We can assume that either $\mathcal{V}(e_1) \Uparrow$ or $\mathcal{V}(e_1) \hookrightarrow^* v$. For the former, clearly $\mathcal{V}(e) \Uparrow$. For the latter, we apply Theorem 1 and deduce that v cannot be a function. Thus our claim holds by SAPPERR1.
  - Wlog assume $\vDash \mathcal{T}((\tau_1 \;@_\tau\; \tau_2))$, $\nvDash \mathcal{T}(\phi_1)$, and $\nvDash \mathcal{T}(\phi_2)$: We know that $\mathcal{V}(e) \hookrightarrow^* ((\mathbf{rec}\ f\ x = e')\ v) \overset{\text{SAPP}}{\hookrightarrow} e''$. By inversion and application of the induction hypothesis our claim holds.

Case $e = \mathbf{match}\ e_0\ \mathbf{with}\ [\overline{C(\overline{x}) \to e'}]$:
  - Wlog assume $\vDash \mathcal{T}(\phi_0)$: By inversion, application of the induction hypothesis, and SMATCHERR our claim holds.
  - Wlog assume $\vDash \mathcal{T}(\phi_p)$: By inversion we get

$$\mathtt{tt}; \tau_0; \Gamma \vdash_p P : \tau_p \;\&\; \phi_p$$

Again by inversion we know that $\mathcal{V}(e_0) \hookrightarrow^* C(\overline{v})$ with $\Gamma \vdash C(\overline{v}) : C(\overline{\tau}) \;\&\; \mathtt{ff}$. Thus we can apply the induction hypothesis and get
    - that there is no matching constructor. Thus

$$\mathcal{V}(e) \overset{\text{SMATCHERR}}{\hookrightarrow^*} \mathcal{V}(\mathbf{match}\ C(\overline{v})\ \mathbf{with}\ [\ ]$$

and by SMATCHNEXTERR our claim holds.
    - that there is a matching constructor. Thus

$$\mathcal{V}(e) \overset{\text{SMATCHNEXT}}{\hookrightarrow^*} \mathcal{V}(e'') \hookrightarrow \mathcal{V}(e') \hookrightarrow^* \mathrm{err}$$

For the pattern-matching cases we have get the following:

Case TPATTERNEMPTY: Trivial follows that there is not matching constructor.

Case TPATTERNNEXT: There are two cases:

- If the first constructor matches the constructor value, then $\nvDash \mathcal{T}(\phi_0)$ because it is the first match and Lemma 5. Furthermore, we have $\vDash \mathcal{T}(C \in \tau_0)$. By assumption we conclude that $\vDash \mathcal{T}(\phi_e)$ and thus apply the induction hypothesis on

$$\Gamma, x_i : \tau_0 {\downarrow}_i^C \vdash e : \tau_e \;\&\; \phi_e$$

  and our claim holds.
- If the first constructor does not match, we apply inversion and the induction hypothesis and on the remaining patterns and get:
    - if there is no matching constructor our claim immediately holds.
    - if there is a matching constructor C_j with body expression e_j and $\mathcal{V}(e_j) \hookrightarrow^* \mathrm{err}$ or $\mathcal{V}(e_j){\Uparrow}$, then our claim holds for $i = j + 1$.