Towards Tree Automata-based Success Types

Robert Jakob and Peter Thiemann

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

Abstract. Error detection facilities for dynamic languages are often based on unit testing. Thus, the advantage of rapid prototyping and flexibility must be weighed against cumbersome and time consuming test suite development. Lindahl and Sagonas' success typings provide a means of static must-fail detection in Erlang. Due to the constraint-based nature of the approach, some errors involving nested tuples and recursion cannot be detected.
We propose an approach that uses an extension of model checking for pattern-matching recursion schemes with context-aware ranked tree automata to provide improved success typings for a constructor-based first-order prototype language.

1 Introduction

Dynamic languages like JavaScript, Python, and Erlang are increasingly used in application domains where reliability and robustness matters. Their advantages lie in rapid prototyping and flexibility. However, no errors are discovered until the erroneous code is actually executed.

Thus, the main error detection facility in dynamic languages is massive unit testing with high code coverage. As the development of unit tests is cumbersome and time consuming, the lack of static analyses that allow error detection prior to execution is one of the major drawbacks of dynamic languages.

Success typings, originally presented by Lindahl and Sagonas [5], provide an approach to statically analyze dynamic languages without losing their benefits: Only mismatches that definitely lead to a type error during execution, are reported. This behavior is different to what a traditional type system provides.

In such a traditional system, the typing $F : \tau_1 \to \tau_2$ means that an application of F to an argument of type τ_1 yields a result of type τ_2 if it terminates normally. However, some programs which do not lead to run-time errors when executed, are rejected by the type system. An example is a conditional that returns values of different types in its branches.

In contrast, a success type system guarantees that for all arguments $v \notin \tau_1$, the function application $F(v)$ leads to a run-time error. For an argument $v \in \tau_1$, success typing gives the same guarantees as traditional typing: $F(v) \in \tau_2$ if it terminates normally. Any approach, however can only approximate the undecidable problem whether a program contains errors.

The system of Lindahl and Sagonas is designed for Erlang and uses a constraint-based algorithm to obtain and refine success types. Their types are drawn from a finite lattice that encompasses atom types and union types. One of the major goals of the original approach was the ability to automatically generate documentation for functions from the inferred success types. This goal requires small, readable types.

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.

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, where the first atom of the named tuple gives the name of the constructor and the other elements are arguments. Thus, the given example corresponds to book("Hamlet","Shakespeare"). Named tuples can be arbitrarily nested and dynamically created.

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.

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

The dialyzer infers the following success type for length:

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

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 that the return value must be either zero or a nested tuple consisting of succ and zero. However, due to approximation, only nestings of three levels are considered. The argument part is exact, i.e. there is no argument of type $[any]$ for which length fails.

To highlight the impression, we create a check function that pattern matches on a suitably nested hierarchy of named tuples. This hierarchy of tuples cannot be created by our length function. Applying our check function to the result of length yields a definite error. However, the success type system in Erlang cannot detect this error.

```
check({succ,{succ,{succ,{foo}}}}) → 0.

test() → check(length([0,0,0,0])).
```

1.2 Our approach

In this paper we focus on these errors and thus only consider programs consisting of trees of constructors as values. Our approach models input type and output type of a function with different models. A success type for a function $f : \mathcal{A} \to \mathcal{G}$ is described with a context-aware ranked tree automaton (caRTA) $\mathcal{A}$ to describe the crash conditions of the function and a pattern-matching recursion scheme (PMRS) $\mathcal{P}$, which essentially is a parametrized pattern-matching tree grammar, to describe the output of the function. Both, automaton and grammar, can accept and construct an infinite tree, respectively.

Our approach yields a PMRS for describing the output with the following rules, where lists are modelled using a nil and cons constructor.

$$\begin{aligned}
&S\ t \to \mathit{Length}\ t\\
&\mathit{Length}\ \mathrm{nil} \to \mathrm{zero}\\
&\mathit{Length}\ (\mathrm{cons}\ x\ xs) \to \mathrm{succ}\ (\mathit{Length}\ xs)
\end{aligned}$$

Here, S is a start symbol which takes an input which can be matched to either nil or cons. We only focus on length's output and ignore the crash conditions. This representation describes the full output length can produce and is no approximation.

To represent the input consumption of the check function presented above, we generate a context-aware ranked tree automaton $\mathcal{A}$ which is able to capture the function's crash behavior. In this simple case, the context-aware ranked tree automaton degenerates to a Büchi tree automaton with a trivial acceptance condition because there are no branches in the control-flow.

$$\begin{aligned}
&\delta(q_{\mathit{Check}}, && \mathrm{succ}) = q_{\mathit{Check}.\mathrm{succ}.1}\\
&\delta(q_{\mathit{Check}.\mathrm{succ}.1}, && \mathrm{succ}) = q_{\mathit{Check}.\mathrm{succ}.\mathrm{succ}.1}\\
&\delta(q_{\mathit{Check}.\mathrm{succ}.\mathrm{succ}.1}, && \mathrm{succ}) = q_{\mathit{Check}.\mathrm{succ}.\mathrm{succ}.\mathrm{succ}.1}\\
&\delta(q_{\mathit{Check}.\mathrm{succ}.\mathrm{succ}.\mathrm{succ}.1}, && \mathrm{foo}) = \epsilon
\end{aligned}$$

Here, q_{Check} is the initial state of the automaton. This automaton only accepts the tree matching the pattern of the check function.

In the example, the length function is called with the list $[0, 0, 0, 0]$. A trivial pattern-matching recursion scheme $\mathcal{G}$ with one rule describes this input:

$$S \to \mathrm{cons\ zero\ (cons\ zero\ (cons\ zero\ (cons\ zero\ nil)))}$$

To check if a definite error occurs, we use an approach given by Ong and Ramsay [6]. This approach allows model checking of the PMRS $\mathcal{P}$ describing the output of the length function, with the tree automaton representing the crash behavior of the check function. For model checking, we require an input to the length function which in our case is $\mathcal{G}$. As expected, the model checking is not successful, because the PMRS $\mathcal{P}$ with the input $\mathcal{G}$ cannot produce the tree required by the automaton $\mathcal{A}$.

Outline The rest of the paper is organized as follows. In section 2 we define pattern-matching recursion schemes and other preliminaries. In section 3 we define a prototype language to demonstrate our approach and in section 4 we transform a given program into a PMRS. We introduce context-aware ranked tree automata for infinite trees, where the predecessors and sibling nodes are considered for transitions, in section 5. Additionally, we outline a transformation from a given program to a caRTA. Section 6 sketches the model checking of a PMRS and a caRTA as an extension of model checking by type inference introduced by Kobayashi and extended by Ong to pattern matching recursion schemes [2, 6], to context-aware ranked tree automata and pattern matching recursion schemes. Section 7 applies this model checking approach to our programs. Finally, we discuss related work in section 8, and conclude in section 9.

2 Preliminaries

This section describes some preliminaries for higher-order and pattern matching recursion schemes. For a detailed introduction please see related work [2, 6].

We define a set of types

$$\tau, \sigma ::= \circ \mid \tau \to \sigma$$

where $\circ$ describes a tree and $\tau \to \sigma$ describes a function from trees of the form τ_1 to trees of the form σ.

We define order and arity of a type as

$$\begin{aligned} &\text{order}(\circ) = 0 && \text{arity}(\circ) = 0 \\ &\text{order}(\tau \to \sigma) = \max\{\text{order}(\tau) + 1, \text{order}(\sigma)\} && \text{arity}(\tau \to \sigma) = \text{arity}(\sigma) + 1 \end{aligned}$$

where order describes the nestedness of the arrow to the left and arity the number of base types $\circ$ on top-level.

A ranked alphabet A is a set of symbols together with arity and order functions defined on each symbol.

With Σ and $\mathcal{N}$ we fix two finite ranked alphabets, where $f, g, h \in \Sigma$ contains only first-order elements called terminal symbols, and $F, G, H \in \mathcal{N}$ contains elements of arbitrary order called non-terminals. Additionally, we define a finite set of variables $x, y, z \in \mathcal{V}$.

Terms created from the just defined ranked alphabets are: Applicative terms $\mathcal{T}(\Sigma, \mathcal{N}, \mathcal{V})$ as expressions built from terminals, non-terminals, and variables using application, patterns $\mathcal{T}(\Sigma, \mathcal{V})$ built by terminal symbols and variables of type $\circ$, and constructor terms $\mathcal{T}(\Sigma)$ built from terminals. Applicative terms must be well-typed with respect to a straightforward simple type system based on types τ.

For a ranked alphabet R we define an R-labelled tree t as a mapping from the string of positive integers $\{1, \dots, n\}^*$ to Σ, with $n = \max\{\text{arity}(s) \mid s \in \Sigma\}$, $\mathbf{dom}(t)$ prefix closed, and if $t(x) = g$, then $\{i \mid x\ i \in \mathbf{dom}(t)\} = \{1, \dots, \text{arity}(g)\}$.

The substitution $a[x/t]$ replaces all occurrences of the variable x by t in the term a. FV(t) denotes the free variables of a term t.

The set of positions Pos of a constructor term $t \in \mathcal{T}(\Sigma, \mathcal{V})$ is defined as:

– If $t = x \in \mathcal{V}$, then $Pos(t) = \{\epsilon\}$.
– If $t = c(t_1, \dots, t_n)$, then $Pos(t) = \{\epsilon\} \cup \bigcup_{i=1}^{n} \{c.i.p \mid p \in Pos(t_i)\}$.

We can transform a position $p \in Pos$ to a string of the alphabet of positive integers by simply omitting the constructor terms c and separator dots, e.g. $c.1.d.2$ is transformed to 12. We use this transformation implicitly. We define a partial order on $p, q \in Pos$ as $p \leq q$ iff there exists p' such that $p.p' = q$. Given a term t and a variable x, $t \searrow_x$ defines the set of all positions of the occurrences of x in t. For a term t and a position $p \in Pos(t)$, we inductively define $t|_p$ as the subterm of t at position p:

$$t|_\epsilon := t \qquad\qquad c(t_1, \dots, t_n)|_{c.i.p} := t_i|_p$$

A *pattern matching recursion scheme* (*PMRS*) [6] is a parametrized grammar for infinite trees. Formally, a PMRS $\mathcal{P} = \langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$ consists of terminal symbols Σ and non-terminal symbols $\mathcal{N}$ as described above, a finite set of rewriting rules, which are either pattern rules or plain rules.

$$F\ x_1 \dots x_n\ p \to t \qquad\qquad F\ x_1 \dots x_n \to t$$

It holds that non-terminal $F \in \mathcal{N}$, variables x_i, terminal patterns $p \in \mathcal{T}(\Sigma, \mathcal{V})$, and a term $t \in \mathcal{T}(\Sigma, \mathcal{N}, \mathcal{V})$ of base type. The start symbol S is a distinguished non-terminal symbol of type $\circ \to \circ$. Additionally, we require that assume that all PMRS we use are well-typed and deterministic, and that $[\![t]\!]_P$ is defined as the value tree of the PMRS P with an argument t according to definitions in [6]. A PMRS P is productive, if $\forall t \in \mathcal{T}(\Sigma). \bot \notin [\![t]\!]_P$. We define $P \downarrow F$ as the PMRS P with the start symbol replaced by F.

A *higher-order recursion scheme* (HORS) is a PMRS without pattern rules.

3 A simple language and its success typings

This section describes a first-order functional language and specifies a notion of success typings for functions defined in this language.

Fig. 1 defines the syntax of the language. A program is a list of function definitions, where all names $(F, G, \dots)$ are different. Functions are unary, as n-ary functions can be emulated with an auxiliary n-ary constructor as a wrapper. Expressions e, r are function applications, constructors $(c, d, \dots)$, where each constructor has a fixed and finite arity, and flat pattern-matching with a variable to match and patterns P. The productions for P define branches for pattern matching in a case expression. The branch for constructor c specifies $n = \text{arity}(c)$ variables and the corresponding expression. There is at most one branch for each constructor in the branches for any case expression in a program. Thus, P can be considered as mapping a constructor to a list of variables and an expression. Finally, values w are trees of constructors v or the special error value err. We do not lose expressiveness by only allowing flat pattern-matching [1]. We require that variables are bound at most once in a program and omit parentheses for nullary constructors.

$$
\begin{array}{ll}
p \ ::= f^* & r \ ::= F(r) \mid c(r_1, \cdots, r_n) \mid x \\
f \ ::= F(x) = e & w ::= v \mid \text{err} \\
e \ ::= \textbf{case}\ x\ \textbf{of}\ P \mid r & v \ ::= c(v_1, \ldots, v_n) \\
P ::= \epsilon \mid c \mapsto (x_1, \ldots, x_n, e), P &
\end{array}
$$

Fig. 1. Syntax definition of our prototype language.

For the transformation of a function to an automaton and a PMRS, we require a special syntactic form. This form ensures that: (1) pattern-matching is only allowed on variables, (2) constructors only contain variables, function applications, or constructors as subterms, and (3) case expressions are not allowed within arguments to function calls. These restrictions can be enforced by preprocessing.[1] Trivial crashes of the form **case** x **of** [] are detected during preprocessing.

The judgement $\Gamma \vdash e \Downarrow w$ defined in fig. 2 describes a big-step semantics for our language. Given an environment Γ mapping from variables x to non-error values v, an expression e evaluates to a value w. For constructor expressions, we evaluate each subexpression and combine the results to a constructor value. If one of the expressions evaluates to an error, the overall result of the constructor expression is an error. A function application is evaluated by evaluating the argument to a value and evaluating the function's body, which we get from an implicit function store, with a new environment where only the argument variable is bound. If the function's argument evaluates to an error, the result of the function application is an error, too. Variables are evaluated by fetching the corresponding value from the environment Γ. There are two rules for case expressions: We fetch the constructor value of the variable to match from the environment Γ and then use the map corresponding to P to get the pattern variables and body e. Then e is evaluated with an environment extended by the variables mapped to values defined in the pattern and constructor argument, respectively. If there is no matching pattern available, we return the error value.

We assume that every program has a function named Main. Thus, our initial judgement for the program evaluation is $\varnothing \vdash \text{Main}(v) \Downarrow w$ with v being an arbitrary non-error value as argument and w being the overall result. Functions may be called recursively.

The only possibility to crash a program is a pattern match failure in a case expression. Thus, the inputs which definitely crash a function only depend on case expressions in the function's body and potential recursive calls. The argument part of a success typing can therefore be formulated as the complement of the inputs which crash the function.

[1] Our goal is to analyze the code, not to run it. In a compiler, such a transformation may not be advisable because it may lead to an exponentially larger program.

$$\textsc{SCtor}\ \frac{\Gamma \vdash e_i \Downarrow v_i}{\Gamma \vdash c(e_1,\dots,e_n) \Downarrow c(v_1,\dots,v_n)} \qquad \textsc{SApp}\ \frac{\Gamma \vdash e \Downarrow v \qquad f(x) = e_b \qquad x \mapsto v \vdash e_b \Downarrow w}{\Gamma \vdash f(e) \Downarrow w} \qquad \textsc{SVar}\ \frac{\Gamma(x) = v}{\Gamma \vdash x \Downarrow v}$$

$$\textsc{SCase}\ \frac{\Gamma(x) = c(v_1,\dots,v_n) \qquad P(c) = (x_1,\dots,x_n,e) \qquad \Gamma, x_1 \mapsto v_1,\dots,x_n \mapsto v_n \vdash e \Downarrow w}{\Gamma \vdash \textbf{case } x \textbf{ of } P \Downarrow w} \qquad \textsc{SCaseErr}\ \frac{\Gamma(x) = c(v_1,\dots,v_n) \qquad c \notin P}{\Gamma \vdash \textbf{case } x \textbf{ of } P \Downarrow \text{err}}$$

$$\textsc{SCtorErr}\ \frac{\exists i. \Gamma \vdash e_i \Downarrow \text{err}}{\Gamma \vdash c(e_1,\dots,e_n) \Downarrow \text{err}} \qquad \textsc{SAppErr}\ \frac{\Gamma \vdash e \Downarrow \text{err}}{\Gamma \vdash f(e) \Downarrow \text{err}}$$

Fig. 2. A big-step semantics for our unrestricted prototype language.

Example 1 (Success Typing). In our prototype language the list length function from the introduction is defined as:

$$\text{Length}(l) = \textbf{case } l \textbf{ of } \text{nil} \to \text{zero},\ \text{cons}(x, xs) \to \text{succ}(\text{Length}(l))$$

The semantics describe that the function crashes on any input that is not a list. The complement of this input is described as

$$\text{nil}, \text{cons}(\top, \text{nil}), \text{cons}(\top, \text{cons}(\top, \text{nil})), \dots$$

and defines a valid argument part for a success typing of Length. Here, $\top$ corresponds to any term. As we already saw in the introduction, the result part of the success typing is any tree consisting of zero and succ. A success typing for length, expressed as a fixpoint of a function on a set of terms, is thus:

$$\mu X.\text{nil} \cup \text{cons}(\top, X) \to \mu Y.\text{zero} \cup \text{succ}(Y)$$

4 Transformation into PMRS

This section gives a transformation of a program p into a PMRS $\mathfrak{T}(p)$, such that both p and $\mathfrak{T}(p)$ generate the same output trees on equal input.

Similar to the transformation given in Kobayashi [2], let **cnames**(p), **fnames**(p), be the ranked alphabets of constructor symbols and function names defined in the program p, respectively. We define a transformation $\mathfrak{T}(p) = \langle \Sigma, \mathcal{N}, \mathcal{R}, \mathcal{S} \rangle$ from programs to PMRS's where $\Sigma =$ **cnames**(p), $\mathcal{N} =$ **fnames**(p), and $\mathcal{S} =$ Main. To get the rules for the PMRS, we transform each function separately $\mathcal{R} = \bigcup \{\mathfrak{T}_{F,x}(e) \mid F(x) = e \in p\}$ where $\mathfrak{T}_{F,a}(e)$ is an auxiliary function from

```
Main(a) =Ack(a)
Ack(a) =  case a of
              pair m n → case m of
                    zero → succ(n)
                    succ x → case n of
                          zero → Ack(pair(x, succ(zero)))
                          succ y → Ack(pair(x, Ack(pair(succ(x), y))))
```

Fig. 3. Definition of the Ackermann function.

function name F, context $a \in \mathcal{T}(\Sigma, \mathcal{V})$ and expression e to a set of rules:

$$\begin{aligned}
\mathfrak{T}_{F,a}(G(e)) &= \{F\ a \to G\ e\}\\
\mathfrak{T}_{F,a}(c(\overline{e})) &= \{F\ a \to c\ \overline{e}\}\\
\mathfrak{T}_{F,a}(x) &= \{F\ a \to x\}\\
\mathfrak{T}_{F,a}(\textbf{case}\ x\ \textbf{of}\ c_i(\overline{y}) \to e_i) &= \bigcup_i \mathfrak{T}_{F,a[x/c_i(\overline{y})]}(e_i)
\end{aligned}$$

In PMRS's, all symbols from Σ and $\mathcal{N}$ are typed. We do not distinguish between different classes of constructors in our PMRS and thus use $\circ$ as single base type. The type for function symbols $F \in \mathcal{N}$ is $F : \circ \to \circ$, as all functions are unary. For constructor symbols $c_i \in \Sigma$ we have $c_i : \underbrace{\circ \to \cdots \to \circ}_{n}$ where n is the arity of the constructor defined in our language.

Example 2 (Transformation of a program to a PMRS). As an example, we transform the Ackermann function into a PMRS. Let p be defined in fig. 3. Using the transformation function $\mathfrak{T}(p)$ we get $\langle \Sigma, \mathcal{N}, \mathcal{R}, \mathcal{S} \rangle$ with

$$\begin{aligned}
\Sigma &= \{\text{pair} : \circ \to \circ \to \circ, \text{succ} : \circ \to \circ, \text{zero} : \circ\}\\
\mathcal{N} &= \{\text{Main} : \circ \to \circ\}\\
\mathcal{S} &= \text{Main}
\end{aligned}$$

and $\mathcal{R}$ as:

$$\begin{aligned}
&\text{Main}\ a = \text{Ack}\ a\\
&\text{Ack (pair zero}\ n) = \text{succ}\ n\\
&\text{Ack (pair (succ}\ x)\ \text{zero}) = \text{Ack (pair}\ x\ \text{(succ zero))}\\
&\text{Ack (pair (succ}\ x)\ (\text{succ}y)) = \text{Ack (pair}\ x\ (\text{Ack (pair (succ}\ x)\ y)))
\end{aligned}$$

For the later defined model-checking approach, we require all PMRS's to be productive. We approximate productivity of a PMRS $\mathcal{P}$ using a standard flow analysis. From now on we assume, that all PMRS's are productive.

5 Transformation into caRTA

This section sketches a transformation of a program into a tree automaton to capture its crash conditions. If the resulting automaton rejects a tree, representing a function's input, then applying the function to this tree yields an error.

Our automata model is a context-aware ranked tree automaton (caRTA), which is an extension of a Büchi tree automaton (BTA). BTA and caRTA differ in their transition function. In a BTA the transition function $\delta_{\mathcal{B}} : Q \times \Sigma \rightarrow Q^*$ describes how to proceed when given a node with a state and a symbol. In a caRTA however, a transition may depend on a finite context of the current node. The context may include ancestors, siblings and their descendants up to a maximum predefined size.

Definition 1 (Context-aware Ranked Tree Automaton). *A context-aware ranked tree automaton (caRTA) $\mathcal{A} = \langle \Sigma, Q, \delta, q \rangle$ is defined by a finite ranked alphabet Σ defining the input symbols, a finite set of states Q, a transition function $\delta : Q \times \mathcal{T}(\Sigma) \times Pos \rightarrow Q^*$ and an initial state q. The transition function maps the current state, a context term, and a path to the current symbol, to a set of states for the children of the current node in the tree. We restrict our transitions such that $\delta(q, t, p) = q_1 \cdots q_n \Rightarrow t|_p = c(x_1, \ldots, x_n)$ where $n = arity(c)$. Furthermore, we do not allow conflicting transitions. Two transitions $\delta(q, t, p) = \overline{q}$ and $\delta(q, t', p') = \overline{q}'$ are conflicting, if t is unifiable with $t'|_{p''}$, where $p' = p''.p$. We define $\mathcal{A} \downarrow q$ as the caRTA $\mathcal{A}$ with the initial state replaced by q.*

A Σ-labelled tree t is accepted by a caRTA $\mathcal{A}$ if there exists a Q-labelled tree r such that both trees have the same domain $\mathbf{dom}(t) = \mathbf{dom}(r)$, and for every $x \in \mathbf{dom}(r)$ there is $\delta(r(x), s, p) = r(x\ 1) \ldots r(x\ m)$ with $m = \text{arity}(t(x))$ and for all $p' \leq p$, $t(x')$ is unifiable with $s|'_p$ and $x = x'p'$. Additionally, the special symbol ? is always accepted by the automaton independent of the current state.

Example 3. We define an automaton $\mathcal{A} = \langle \Sigma, Q, \delta, q_0 \rangle$ with

$$
\begin{array}{ll}
\Sigma = \{a : \circ \rightarrow \circ \rightarrow \circ, b : \circ, c : \circ\} & \\
Q = \{q_0, q_1, q_2\} & \\
\delta(q_0, \underline{a}, \epsilon) = q_1\ q_2 & \delta(q_*, \underline{*}, \epsilon) = \overline{q_*} \\
\delta(q_1, a\ \underline{b}\ x, a.1) = \epsilon & \delta(q_2, a\ b\ \underline{*}, a.2) = \overline{q_*} \\
\delta(q_1, a\ \underline{c}\ x, a.1) = \epsilon & \delta(q_2, a\ c\ \underline{c}, a.2) = \epsilon
\end{array}
$$

With $*$ we represent that any term may occur at this position. The state q_* is a drain state that accepts any term The current node of the transition which corresponds to the path is emphasized by underlining. The automaton accepts the following trees:

$$\forall t \in \mathcal{T}(\Sigma).a\ b\ t \text{ and } a\ c\ c$$

The transformation from a program p to a caRTA $\mathcal{A}$ proceeds in three steps: At first, we translate the different pattern-matching cases into transitions of an automaton. Then, we analyze the function calls and the variable bindings and extract constraints for the variables. Finally, we transform the automaton according to the constraints.

$$\mathfrak{S}_{V,\sigma,F,a}(r) = \bigcup_{v \in V} \{(\sigma(v), a', pos) \to \overline{q_*}\}$$

where:

$$a' = a[*]_{a \searrow v}$$
$$pos = a \searrow v$$

$$\mathfrak{S}_{V,\sigma,F,a}(\textbf{case } x \textbf{ of } c_i(\overline{x_i}) \to e_i) = \bigcup_i \delta_i \cup \delta'$$

where:

$$\mathfrak{S}_{V_i,(\sigma,x_{i1} \mapsto q_{p_{i1}},\ldots,x_{im} \mapsto q_{p_{im}}),F,a_i}(e_i) = \delta_i$$
$$a_i = a[x/c_i(\overline{x_i})]$$
$$V_i = V \setminus x \cup \{\overline{x_i}\}$$
$$p_{ij} = a_i \searrow_{x_{ij}}$$
$$\delta' = \bigcup_i \{(\sigma(x), a_i, a \searrow_x) \to q_{p_{i1}} \ldots q_{p_{im}}\}$$

Fig. 4. Transformation from an expression to a set of caRTA transitions.

5.1 First step: Creation of the automaton

As the only possibility to crash a program is a pattern match failure, the first step only transforms the nested pattern-matching structure of the given program and its functions into a caRTA. The contexts of the caRTA are used to map the control-flow of the program into the automaton.

We use a transformation $\mathfrak{S}(p) = \langle \Sigma, Q, \delta, q \rangle$ from a program p to a caRTA, where $\Sigma = \textbf{cnames}(p)$, $Q = \textbf{qnames}(\delta)$ which extracts all states from the transitions in δ, and $q = q_{\text{Main}}$. We define $\delta = \bigcup \{\mathfrak{S}_{\{x\},x \mapsto q_F,F,x}(e) \mid F(x) = e \in p\}$, which calls the auxiliary function $\mathfrak{S}_{V,\sigma,F,a}(e)$ defined in fig. 4 to transform pattern-matching cases into transitions. The auxiliary function $\mathfrak{S}_{V,\sigma,F,a}(e)$ maps from a set of variables V, which contains the variables that have not been pattern-matched so far, an environment σ as mapping from variable to state, a function symbol F, the current context $a \in \mathcal{T}(\Sigma, \mathcal{V})$ and the current expression e to a set of transitions. For each pattern in a case expression, a transition rule is created using the current context. The auxiliary function is applied on every pattern body with adjusted arguments and the resulting transitions are collected. The set of the not-pattern matched variable set is adjusted, and the context is adapted because we gain information about variables through pattern matching. Additionally, the environment σ is extended with variables defined in the patterns. The states are created using the paths to the variables in the new context. When an expression r is encountered, no more case expressions can occur. Thus, we create transitions into drain states for all variables in V.

We remember the environment σ_p which maps all variable in p to their corresponding state. The context a is necessary, because the automaton has to respect control flow in the program.

Example 4 (Creation of automaton). We apply $\mathfrak{S}(p)$ to the Ackermann function defined in example 2 and obtain $\mathfrak{S}(p) = \langle \Sigma, Q, \delta, q_{\text{Main}} \rangle$ as defined in fig. 5.

Function calls are not considered in this step, thus for all variables which are not pattern-matched, drain transitions are created.

$$
\begin{aligned}
&\Sigma = \{\mathrm{pair} : \circ \rightarrow \circ \rightarrow \circ, \mathrm{succ} : \circ \rightarrow \circ, \mathrm{zero} : \circ\} \\
&Q = \{q_{\mathrm{Main}}, q_{Ack}, q_{Ack.pair.1}, q_{Ack.pair.2}, q_{Ack.pair.1.succ.1}, q_{Ack.pair.2.succ.1}\} \\
&\delta(q_{\mathrm{Main}}, \underline{*}, \epsilon) = \overline{q_*} \\
&\delta(q_{Ack}, \underline{\text{pair m n}}, \epsilon) = q_{Ack.pair.1}\ q_{Ack.pair.2} \\
&\delta(q_{Ack.pair.1}, \text{pair } \underline{\text{zero}}\ n, \text{pair.1}) = \epsilon \\
&\delta(q_{Ack.pair.2}, \text{pair zero } \underline{*}, \text{pair.2}) = \overline{q_*} \\
&\delta(q_{Ack.pair.1}, \text{pair } (\underline{\text{succ } x})\ n, \text{pair.1}) = q_{Ack.pair.1.succ.1} \\
&\delta(q_{Ack.pair.1.succ.1}, \text{pair } (\text{succ } \underline{*})\ n, \text{pair.1.succ.1}) = \overline{q_*} \\
&\delta(q_{Ack.pair.2}, \text{pair } (\text{succ } x)\ \underline{\text{zero}}, \text{pair.2}) = \epsilon \\
&\delta(q_{Ack.pair.2}, \text{pair } (\text{succ } x)\ (\underline{\text{succ } y}), \text{pair.2}) = q_{Ack.pair.2.succ.1} \\
&\delta(q_{Ack.pair.2.succ.1}, \text{pair } (\text{succ } x)\ (\text{succ } \underline{*}), \text{pair.2.succ.1}) = \overline{q_*}
\end{aligned}
$$

Fig. 5. The caRTA for the Ackermann function after the first step.

$$
\begin{aligned}
\mathcal{Q} = \{&(q_{Ack.\mathrm{pair}.1.succ.1}, q_{Ack.\mathrm{pair}.1.succ.1}), \\
&(q_{Ack.\mathrm{pair.1.succ.1}}, q_{Ack.\mathrm{pair.1}}), \\
&(q_{Ack.\mathrm{pair.2.succ.1}}, q_{Ack.\mathrm{pair.2}})\}
\end{aligned}
$$

Fig. 6. Non-closed relation $\mathcal{Q}$ on states for the Ackermann function.

5.2 Second step: Analyzing the calls

The automaton we just created has to be adapted in a second step as it does not respect constraints induced by function calls. Each variable is represented as a state in our automaton according to the mapping σ_p. Thus, we analyze use of variables in each function call to detect equivalent states.

We define $\mathcal{Q} : Q \times Q$ to be the smallest relation closed under reflexivity, symmetry, and transitivity with:

$$\mathcal{Q} = \{(\sigma_p(x), q_{F.s})\} \mid F(e) \in p, x \in FV(e), s \in e \searrow_x\}$$

Thus, $\mathcal{Q}$ is an equivalence relation created from all variables used as arguments to function applications in the program. The current state of the variable is obtained from the previously defined mapping σ_p. The new state the variable should be mapped to, is built by extracting the path to the variable in the argument, prepended by the name of the function that is called.

Example 5 (Call analysis). The Ackermann function has three function calls where four variables are used. The equivalence relation is given in fig. 6.

$$
\begin{aligned}
&\delta(q_{Ack}, \underline{\text{pair m n}}, \epsilon) = q_{Ack.pair.1}\ q_{Ack.pair.2} \\
&\delta(q_{Ack.pair.1}, \text{pair } \underline{\text{zero}}\ n, \text{pair.1}) = \epsilon \\
&\delta(q_{Ack.pair.2}, \text{pair zero } \underline{*}, \text{pair.2}) = \overline{q_*} \\
&\delta(q_{Ack.pair.1}, \text{pair } (\underline{\text{succ } x})\ n, \text{pair.1}) = q_{Ack.pair.1'} \\
&\delta(q_{Ack.pair.2}, \text{pair (succ } x)\ \underline{\text{zero}}, \text{pair.2}) = \epsilon \\
&\delta(q_{Ack.pair.2}, \text{pair (succ } x)\ (\underline{\text{succ } y}), \text{pair.2}) = q_{Ack.pair.2'} \\
&\delta(q_{Ack.pair.1'}, \text{zero}, \epsilon) = \epsilon \\
&\delta(q_{Ack.pair.1'}, \text{succ } x, \epsilon) = q_{Ack.pair.1'} \\
&\delta(q_{Ack.pair.2'}, \text{zero}, \epsilon) = \epsilon \\
&\delta(q_{Ack.pair.2'}, \text{succ } x, \epsilon) = q_{Ack.pair.2'}
\end{aligned}
$$

Fig. 7. The final automaton capturing the crash conditions for the Ackermann function.

5.3 Third step: Intersection of the automaton

Finally, we have to adapt the automaton created in the first step to the state equivalence relation created in the second step.

This last step is surprisingly difficult because of the necessity to retain the contexts such that the original control-flow is unharmed. We are working on a formalization for this step.

Example 6 (Final automaton). For the Ackermann function, a possible final automaton is defined in fig. 7. This automaton can be minimized, as the transitions for $q_{Ack.pair.1'}$ and $q_{Ack.pair.2'}$ are equivalent.

6 Model checking

This section discusses the question whether the output trees from a PMRS $\mathcal{P}$ with some input defined by a HORS G are accepted by a caRTA $\mathcal{A}$.

As we do not know the input to our functions in general, we use a trivial input HORS $\mathcal{G}_?$ defined as

$$
\begin{aligned}
&\mathcal{G}_? = \langle \Sigma = \{?\}, \mathcal{N} = \{S\}, R, S \rangle \\
&\text{with } R : S \to ?
\end{aligned}
$$

As defined above, ? is a special symbol always accepted by an arbitrary caRTA.

Given a deterministic PMRS $\mathcal{P}$, the HORS $\mathcal{G}_?$ just defined, and a context-aware ranked tree automaton $\mathcal{A}$, we define:

$$\models (\mathcal{P}, \mathcal{A}) \quad \text{iif} \quad \forall t \in \mathcal{L}(\mathcal{G}_?).\ [\![Main\ t]\!]_{\mathcal{P}} \in \mathcal{L}(\mathcal{A})$$

The *caRTA-extended PMRS verification problem* is to decide the truth of $\models (\mathcal{P}, \mathcal{A})$.

Our problem is similar to the PMRS verification problem described by Ong and Ramsay [6]. We can follow their approach based on a counter-example guided abstraction refinement (CEGAR) until the last step: At first, an over-approximation of the given PMRS is calculated. Then, they eliminate non-determinism by adding a family of terminal symbols, symbolizing the non-deterministic choice and adopt the automaton. This adoption is possible on our caRTA, too. Finally, they end up with recursion schemes with weak-definition-by-cases (wRSC), which are equi-expressive to the over-approximated PMRS.

The model checking of the wRSC with the automaton $\mathcal{A}$ is based on a type inference. We have to extend this type inference because we have to cope with the contexts present in our context-aware ranked tree automaton.

6.1 Model checking by type inference

Model checking by type inference [6] is based on an intersection type system.

We plan on extending this type system as follows: Let n be the maximum height of the contexts used in $\mathcal{A}$. We required our initial PMRS to be productive, thus, wRSC is productive, too. Then, from the wRSC $\mathcal{G}$, we approximate the set of trees of height n that occur in the value tree of $\mathcal{G}$ as T. Finally, we have to adapt the TERM rule from the intersection type system such that it checks whether the contexts requested from the automaton occur in T.

7 Checking programs for definite errors

This section sketches an idea how to find definite errors in a given program p using the model checking possibilities described above.

From our program p we create a PMRS $\mathfrak{T}(p)$ and a caRTA $\mathfrak{S}(p)$. For every function application $F(t)$ we check for definite errors

- If the argument contains no function application, we check if if $\mathfrak{S}(p) \downarrow q_F$ accepts by the argument t where all variables are replaced by ?.
- If the argument contains $(G(t')$ with a possible prefix c, we insert a new rule $F'{}_{\_} \rightarrow t$ and verify that $\vDash (\mathfrak{T}(p) \downarrow G, \mathfrak{S}(p) \downarrow q'_F)$ holds.

8 Related work

Lindahl and Sagonas [5] have proposed success types out of the consideration that standard type systems do no provide useful guarantees for dynamically typed languages. They have defined and implemented a widely used inference algorithm that is based on constraint solving using fixpoint iteration. Subsequently, they have considered the interplay of success types with contracts and suggested a slicing-based algorithm for improved error reporting [7].

Suter and coworkers [8] introduce an extension of Scala with statically checked contracts which are provided by the programmer. These contracts may include recursively defined functions and thus may be used for stating type refinements.

The contribution of the paper is the SMT-solver-based contract checking procedure. In our work, we are interested in inferring refinements in the form of success types.

Our approach draws a lot of inspiration from work on higher-order model checking by Kobayashi and coworkers [4, 2, 3] as well as by Ong and coworkers [6]. We are essentially applying the procedure developed by Kobayashi [3] to a generalized automata model, the caRTA introduced in this paper. The goal of the generalization is to obtain more precise analysis results; the price is that our analysis is restricted to productive systems, which may exclude certain programs from the analysis. We need to gather further experience with our prototype implementation to judge the impact of this restriction.

9 Conclusion

We propose a new approach for inferring success typings that uses context-aware ranked tree automata (caRTS) and pattern-matching recursion schemes (PMRS) as models to obtain more precise results than the algorithm of Lindahl and Sagonas. We outline a transformation from a first-order functional language to caRTS and PMRS and sketch an extension of model checking by type inference for these models.

References

1. L. Augustsson. Compiling pattern matching. In *Proc. FPCA 1985*, volume 201 of *LNCS*, pages 368–381. Springer, 1985.
2. N. Kobayashi. Model-checking higher-order functions. In A. Porto and F. J. López-Fraguas, editors, *PPDP*, pages 25–36. ACM, 2009.
3. N. Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In B. Pierce, editor, *POPL*, pages 416–428, Savannah, GA, USA, Jan. 2009. ACM Press.
4. N. Kobayashi and A. Igarashi. Model-checking higher-order programs with recursive types. In M. Felleisen and P. Gardner, editors, *ESOP*, volume 7792 of *Lecture Notes in Computer Science*, pages 431–450. Springer, 2013.
5. 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.
6. C.-H. L. Ong and S. J. Ramsay. Verifying higher-order functional programs with pattern-matching algebraic data types. In *POPL*, pages 587–598, Austin, TX, USA, Jan. 2011. ACM Press.
7. 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.
8. P. Suter, A. S. Köksal, and V. Kuncak. Satisfiability modulo recursive programs. In E. Yahav, editor, *Static Analysis - 18th International Symposium, SAS 2011*, volume 6887 of *Lecture Notes in Computer Science*, pages 298–315, Venice, Italy, Sept. 2011. Springer.