A Success Typing System on Imperative Programming Language

Yishan Zhao
University of Maryland
yishanzh@umd.edu

Abstract

Type system is not a foreign concept to programmers, being present even in some of the most dynamically typed languages like Javascript. A typical language that fully utilizes the concept of types would require a type definition in order to instantiate a variable of that type. Or in some extreme scenarios, patching up concepts like interfaces, abstract classes and even various design modes to impose constraints on a variable's type. Creating a variable without consciously thinking about its type seems like a liberty reserved to insecure dynamically typed languages. Although this paper is not a one-size-fits-all solution to an expressive, minimalistic yet rigorous type system, it presents one that omits the necessity of explicitly defining any type, but at the same time deterministic and statically decidable. this type system is similar to that of Erlang's, but with a few major differences. First, values are mutable and as a result, parts of the typing rules are generalized to facilitate the extra expressiveness that it brings. Second, array types are defined with bounds, which adds additional safety to indexing operations. Finally, the type system introduces the concept of constraints to provide further safety checks and narrow down the set of false-negative results during type-checking. In addition, this paper will also propose and discuss solutions to the unsolved issues of the core Erlang success typing.

1. Introduction

Type inference is not a new concept. In many mainstream functional programming languages, programmers can enjoy the benefit of connecting pieces of the program together without ever having to write down a single type signature. Yet, most if not all languages still retain the notion of declaring an algebraic type so that members of this type can be constructed. Could a sufficiently sophisticated type inference algorithm get rid of type definitions at all, so that every expression can be automatically matched to the most suitable type? When experienced programmers find themselves reviewing the other one's code without proper documentation on every variable or function, normally their reaction would be to guess the properties of said variables without resorting to documentation. In short, terms and expressions are usually understood by observing their behaviors and properties under various conditions, rather than studying the ground truth, i.e. the type.

This is the heuristic of this type system: If a term exhibits observable properties, then it must be a subtype of its "true" type. Every new bit of information from type inference helps to narrow down the subtype even further, to a point where all necessary type information about this term has been discovered or the final subtype turns out to be $\perp$, a contradiction thus arises and typechecking concludes.

In such a system, the typechecker must be optimistic about the expressions and statements [4]. In other words, there is an underlying assumption that everything in this program is correct and introduces new knowledge to the types, unless there is no way to reconcile the emerging contradictions. At that point, the program must fail with no exceptions.

It is worth noticing that the aforementioned strategy is exactly the same as in success typing. In essence, the typechecker optimistically assumes that the program is correctly typed, until an indisputable contradiction arises. This behavior will be discussed in more detail in the next section.

2. Related Works

Despite the original exploration focus on a better type inference algorithm and minimal type annotation, it naturally and inevitably leads to success typing.

2.1. Success Typing

The idea of success typing is developed from soft typing. Soft typing is one of the earlier works by Cartwright, et. al[2] attempting to generalize dynamic typing and static typing together into one single framework. By default, the typechecker accepts all valid programs without type annotation to serve the purpose of softly transforming a dynamically typed language into a statically typed one through the process of annotating and type inference. Inspired by soft typing, success typing in Erlang (Lindahl, et. al[6]) inher-

its these traits and then allows types to be composed in a bottom-up fashion in order to scale well in practice.

2.2. Conditional Typing

The type system in Erlang does take control flows into consideration when solving for the type of an expression [6]. However, upon closer inspection, these additional typing rules are only introduced by pattern matching and in short, the inferred type of a case expression is always the union type of all its branches. While this approach is safe and practical, pattern matching is only one of the many branching operations. Another prior work by Aiken et. al[1] on combining soft typing with conditional types reaches similar approaches, but with less expressiveness due to the lack of concepts akin to Erlang's constraints.

A work from 2011, by Guha et. al[3] also tries to leverage the power of flow control analysis, but the scope is limited to developing a compatible type system for JavaScript, which greatly limits the space for type annotation and type inference.

3. Language and Typing Rules

After decades of industrial application, most if not all of the current imperative languages are complete with sophisticated syntax and a diversity of practical features. This language does not aim to compete in terms of completeness nor practicality, but rather a demonstration of this improved success type system.

This language contains only the minimum features of a typical imperative language, namely conditional branches, iterative loops, simple data structures and ways to alter the values of variables.

3.1. Programming Language Syntax

The syntax allows for the minimum set of instructions that any non-trivial and Turing Complete language must have, shown in 1. The goal of this minimalism is to remove unnecessary features that could complicate the type system by introducing irrelevant edge cases that would require extra proof.

The control flow consists of the `if` and `while` structures that behave exactly like in many high-level imperative programming languages, such as Java. The `return` and `break` statements are implemented as propagating termination flags through the stack within the interpretation process. A noticeable trait of this language, or rather the lack thereof, is the tuples. Arrays in this language can have elements of different types, thus virtually marks the concept of tuples redundant.

There are a few extra notations that are worth explanation.

- $\circ$ is the associative operator that joins two strings together.
- $\{\overline{x} : \overline{e}\}$ is the struct literal with a list of key-value pairs, with the keys denoted as $\overline{x}$ and values $\overline{e}$.
- Arrays have two constructors: they are either from an array literal $[e_1, \ldots, e_n]$, or in a C-like style where the length and type are given, i.e. $T[e]$.
- The assignment expression is a bit different: the variable x could be an indexing on an array (in the form of $x[i]$ where i is a natural number), or a projection (in the form of $x.y$). While it is possible for this syntax definition to give rise to assignments such as $\{x : 1\}.x = 2;$ where values are assigned to constant left-hand-side expressions, these do not have any effect on the runtime environment.

The operational semantics for this syntax virtually follow the usual expectation for imperative languages of this kind. However, the indexing and projection slightly deviate from a typical definition, in that if the referred key or index does not exist, they will be declared at that point and appended to the corresponding data structure. This design choice may seem counter-intuitive, but it is due to the peculiarities of success typing that will be discussed in detail in the next subsection.

It is worth mentioning that the `switch` is rather different from expectation, as it is instead a pattern matching. Normally, the case expression matches the constructors of an existing type. As no types are explicitly defined in this language, the only constructors that it can match are from structure and array literals. It may appear quite limiting on the expressiveness, but considering that tuples are equivalent to short arrays and singleton types allow the creation of singleton types to simulate the behavior of type constructors, the `switch` statement is not as limiting as it appears to be. For example, consider the following:

```
sum(tree) {
    switch (tree) {
        case "Leaf":
            return 1;
        case ("Branch", l, r):
            return sum(l) + sum(r);
    }
}
```

tree can be thought of as a member of the Tree type that has two constructors, defined as follows:

$$\begin{aligned} Leaf &: \text{Tree} \\ Branch &: \text{Tree} \to \text{Tree} \to \text{Tree} \end{aligned}$$

It may appear confusing that, there is no guarantee that the tree variable is indeed the *exact* type, it could very possibly have another constructor that is not matched by the case expression. The concern is indeed valid, but the primary

$$
\begin{aligned}
s ::= & \; x \;=\; e \\
& \mid \texttt{if (}e\texttt{) \{}e_1\texttt{\} then \{}e_2\texttt{\}} \\
& \mid \texttt{while (}e\texttt{) \{}e_1\texttt{\}} \\
& \mid \texttt{return;} \\
& \mid \texttt{break;} \\
& \mid e(\overline{e'}) \\
& \mid f(\overline{x})\{\overline{s}\} \\
& \mid \texttt{switch (}e\texttt{) \{}p_1 \to \overline{c_1} \;\ldots\; p_n \to \overline{c_n}\texttt{\}} \\
e ::= & \mid e_1 \;\leq\; e_2 \mid e_1 \;\geq\; e_2 \\
& \mid e_1 \;\wedge\; e_2 \mid e_1 \;\vee\; e_2 \mid \neg e \\
& \mid e_1 \;+\; e_2 \mid e_1 \;-\; e_2 \\
& \mid e_1 \circ e_2 \\
& \mid \text{int} \mid \text{bool} \mid \text{char} \\
& \mid x \mid e.y \mid e[e] \mid f(\overline{e}) \\
& \mid [e_1, \ldots, e_n] \mid T[e] \\
& \mid \{\overline{x} : \overline{e}\}
\end{aligned}
$$

Figure 1. The operational semantics of this language, omitting the optional type annotations.

$$
\begin{aligned}
\tau ::= & \; \tau_1 \cup \tau_2 \\
& \mid \text{int} \mid \text{bool} \mid \top \mid \bot \mid \circ \\
& \mid \langle \tau,\; n,\; \overline{i} : \overline{\tau_i} \rangle \\
& \mid \{\overline{x} : \overline{\tau_x}\} \\
& \mid \overline{\tau} \mapsto \tau' \\
& \mid \{1\} \mid \{2\} \mid \ldots \\
& \mid [n_1,\; n_2) \\
& \mid \{true\} \mid \{false\}
\end{aligned}
$$

Figure 2. The types

focus of success typing is not to capture the lower bound of the types — that would be the definition of static typing — but rather to set an upper bound for every type so that programs that *will* fail can be identified.

3.2. Type

The type system is quite similar to Erlang's. However, the presence of mutable variables introduces challenges and necessary generalizations to the typing rules. For example, the following code is perfectly valid in this type system:

```
x = true;
x = 1;
```

The underlying reason and corresponding typing rules will be discussed in the next section.

Although the goal of this type system is to omit type declarations completely, it is undeniable that appropriate type annotation is still crucial to a readable and secure program. Therefore the typing rules are designed in a way that is compatible with type annotations. However, they will not be shown in this paper since the work is rather trivial.

3.3. Success Typing and mutable variable

A natural generalization of the original success typing would be that a term can only be assigned to a variable if the term is a subtype of that variable, i.e.

$$
\frac{A \cup \{x \mapsto \tau'\},\; C \vdash e : \tau \subseteq \tau'}{A,\; (C - \overline{C_x}) \cup C[e \mapsto x] \vdash x = e : \circ, \quad x : \tau}
$$

The advantage is obvious: the type of any term will narrow down monotonously, thus if the said term terminates, it must preserve all the previously known typing relations, regardless type inference reaches $\bot$. An additional benefit is that in a `while` loop, types of all terms can always be guaranteed to be well-defined without the extra need to backtrack. Although this is a plausible axiom, it offers little freedom to the semantics. Consider this example:

```
x = 5;
x = 6;
```

Line 1 assigns 5 to x, so that the type of x is the singleton type $\{5\}$. However, $\{5\}$ is not a subset of $\{6\}$ thus the second line will reach a type error. Considering this type of assignment is extremely common, it would be ridiculous to prohibit it. Thus the current approach is to allow both union and intersection to be valid algebra. However, the loop structure becomes, once again, the critical issue. For example,

```
y = [...]; /* a bunch of values
that share little in common */
i = 0;
x = 0;
while(b) {
    x++;
    x = y[i];
    i++;
}
```

The type of x can grow arbitrarily large: suppose $x : X_0$ during the first iteration, in the second iteration, $x : X_1 = X_0 \cup Y_i$ where Y_i is the type of the i^{th} element, that does not necessarily have to be a subset of X_0, and therefore break the typing relation. This has the disastrous implication that, if the typechecker naively prohibits backtracking, the only

$$\frac{}{A \cup \{x \mapsto \tau\},\ C,\ S \vdash x : \tau \quad V = \emptyset}\ \text{T-Var}$$

$$\frac{A,\ C \vdash x : \tau = \langle \tau,\ n,\ \ldots \tau_i \ldots \rangle,\ i : \tau_j}{A,\ C \cup \overline{C_i} \vdash x[i] : \tau_i, \quad i : \tau_j \cap [0, n) \quad V = \emptyset}\ \text{T-Index}$$

$$\frac{A,\ C \vdash x : \tau \cup \{x.y \mapsto \tau'\}}{A,\ C \vdash x.y : \tau' \quad V = \emptyset}\ \text{T-Proj}$$

$$\frac{A, C \vdash e_1 : \tau_1,\ \ldots,\ e_n : \tau_n \quad V = \overline{v_{e_i}}}{A, C \vdash c(e_1,\ \ldots,\ e_n) : c(\tau_1,\ \ldots,\ \tau_n) \quad V = \overline{v_{e_i}}}\ \text{T-Struct}$$

$$\frac{A,\ C \vdash e_1 : \tau_1,\ \ldots,\ e_n : \tau_n \quad V = \overline{v_{e_i}}}{A,\ C \cup \overline{C_i} \vdash [e_1,\ \ldots,\ e_n] : \langle \tau,\ n,\ \overline{\tau_i} \rangle \quad V = \overline{v_{e_i}}}\ \text{T-Arr1}$$

$$\frac{A,\ C \vdash e : \tau_e \quad V = \overline{v_e}}{A,\ C \vdash \tau[e] : \langle \tau,\ n,\ \emptyset \rangle, \quad e : \tau'_e = \text{glb}\ (\tau_e, \mathbb{Z}), \quad n = \max\ \tau'_e \quad V = \overline{v_e}}\ \text{T-Arr2}$$

$$\frac{A,\ C \vdash e : \tau \quad V = \overline{v_e} \qquad A \cup \{x \mapsto \tau\}, C \vdash x : \tau}{A,\ (C - \overline{C_x}) \cup C[e \mapsto x] \vdash x = e : \circ, \quad x : \tau \quad V = \overline{v_e} \cup \{x\}}\ \text{T-Assign}$$

$$\frac{A \cup \{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n, f \mapsto \tau_f\},\ C \vdash e : \tau_e \quad V = \overline{v_e}}{A,\ C \vdash \tau_f\ f(x_1, \ldots x_n) \to e : \circ, \quad \tau_f = ((\tau_1, \ldots, \tau_n) \to \tau_e) \quad V = \overline{v_e}}\ \text{T-Abs}$$

$$\frac{A,\ C \vdash f : \tau_f, e_1 : \tau_1, \ldots, e_n : \tau_n \quad V = \overline{v_{e_i}}}{A,\ C - \overline{C_x} \vdash f(e_1, \ldots, e_n) : \beta, \quad \tau_f = \alpha_1, \ldots, \alpha_n \to \alpha, \quad \beta \subseteq \alpha, \quad \tau_1 \subseteq \alpha_1, \ldots, \tau_n \subseteq \alpha_n \quad V' = V_f \cup \overline{v_{e_1}}}\ \text{T-App}$$

$$\frac{\begin{array}{c} A \cup \{v \mapsto \tau_v | v \in Var(p_1)\},\ C \vdash p_1 : \alpha_1,\ b_1 : \beta_1 \\ \vdots \\ A,\ C \vdash e : \tau,\ C_e \quad A \cup \{v \mapsto \tau_v | v \in Var(p_n)\} \vdash p_n : \alpha_n,\ b_n : \beta_n \quad V = \overline{v_e} \end{array}}{A,\ C \vdash (\texttt{switch}\ (e)\ p_1 \to b_1; \ldots; p_n \to b_n) : \circ \quad V = \overline{v_e} \cup \overline{v_{b_i}}}\ \text{T-Case}$$

$$\frac{\begin{array}{c} A,\ C \vdash c \in \text{bool} \quad V_c = \overline{v_c} \\ A,\ C \cup C_t - C_{V_c} \vdash,\ e_t : \tau_t \quad V_t = V_c \cup \overline{v_{e_t}} \\ A,\ C \cup C_f - C_{V_c} \vdash e_f : \tau_f \quad V_f = V_c \cup \overline{v_{e_f}} \end{array}}{A,\ C \vdash (\texttt{if}\ (c)\ e_t\ e_f) : \circ \quad V = V_t \cup V_f}\ \text{T-If}$$

$$\frac{\begin{array}{c} A,\ C \vdash c \in \text{bool} \quad V_c = \overline{v_c} \\ A,\ C \cup C - C_{V_c} \vdash,\ e : \tau \quad V = V_c \cup \overline{v_e} \end{array}}{A,\ C \vdash (\texttt{while}\ (c)\ e) : \circ \quad V = V_c \cup \overline{v_e} \quad \forall v \in V. A[v/v'](v) \subseteq A(v)}\ \text{T-While}$$

Figure 3. The majority of the typing rules, note that the rule T-App is drastically different from a normal one

sensible type to assign to x is at most X_1 which contradicts the typing rule for `x++` during the second iteration.

Backtracking seems crucial to the solution, but it is at the cost of determinism which is also an indispensable property for a practical typechecker. Thus the compromise is to use a special typing rule for a loop, so that all terms at the end of an iteration must be a subtype of what they start to be. To keep track of this without backtracking, in Fig. 3, there is a set V being the set of all variables whose states may be altered in the current scope and up to this point. Similarly,

the notation v_e denotes all the variables that may be altered in the expression e. In addition, the set Valso helps to determine whether a constraint in set C is "outdated", meaning the constraint no longer applies to some variables' states.

3.4. Bounded Array

As mentioned above, some types can be dependent on statically determined values, namely the types with the singleton property. It is natural to ask, are there any more types that can be constructed in this way? Admittedly, constructions in the form of $\mathrm{N} \to \tau$, where N is the set of all normal forms, make sense in only a few selected contexts, but a bounded array would be such one.

Array has been plagued by the issue of invalid indices for a long time, and there are numerous attempts to reconcile the intrinsic conflict between unsafe memory access and the convenience that it provides.

Arrays in this language is encoded by a triplet, $\langle \tau,\ n,\ \overline{(i,\ \tau_i)} \rangle$, where τ represents the lowest upper bound of the types of all its elements, $n \in \mathbb{N}$ is its length and $\overline{(i,\ \tau_i)}$ is the set of scoped type relations, that the element at index i is not only a subtype of τ, but is also a subtype of τ_i.

This notation may appear unintuitive at first glance, but it is a compromise between expressiveness, practicality in terms of memory consumption and convenience. If an array is constructed from the expression $\tau[n]$, then n can be arbitrarily large, while all elements are of exactly the same type. On the other hand, if it comes from an array literal, n should be relatively small, although the elements might have drastically different types.

However, if n is statically determined, how could the type express dynamically bounded arrays? Consider this example:

```
x = scan();
```

Suppose scan is an IO action that returns a dynamically allocated array of characters, which are then assigned to x. What would the type of x be? Since there are no further information about the size of this array, it is only sensible to assume the length to be maximum integer. Admittedly, this is a rather peculiar and useless choice: it does not enforce static check on bounds, which is the sole purpose of type checking [5]. However, this is due to the nature of success typing that a typechecker does not signal false positives, but rather raises a contradiction only when the program is destined to fail.

We can, however, make this system more useful than it first appears. Consider the following:

```
y = some_int;
x = int[42];
... = x[y];
b = y <= 42;
```

x is an array of length 42, and then it is indexed by y. From line 1, the typechecker infers that y is a subtype of the integer, $[\mathrm{int.min},\ \mathrm{int.max}]$. The typechecker then optimistically assumes the indexing will happen at line 3 is the intended behavior, not contradicting against any known type relations. In order for it to be true, y has to be lowered to the subtype of $[0,\ 42)$. Finally, this additional information helps to determine that b has a more specific type, i.e. the singleton type $\{false\}$.

As mentioned above, this inference is only possible if the typechecker keeps updating the existing knowledge of the types. On the other hand, if the newly inferred information produces a contradiction with existing type relations in the form of a bottom type, then it is an indirect way of warning the programmer that the assumption about arrays' bounds may be flawed.

3.5. Flow-sensitive Typing

Additionally, we can make the claims on bounded arrays even stronger by combining it with flow-sensitive typing. Consider this example:

```
if (a < 42) {
    x = int[a]
}
```

Intuitively, we would like the typechecker to infer that x can only be of type $\langle \mathrm{int},\ a,\ \emptyset \rangle$. However, knowing that the bounds are statically determined, the best guess can only be that $x : \langle \mathrm{int},\ 42,\ \emptyset \rangle$. To obtain this crucial constraint, the typechecker must allow boolean expressions to be used as if it is a relation on variables. In an ideal setting where values are immutable and no side-effects are involved, we can safely assume that any binary operation that maps to a boolean value is a relation defined on the two types.

While $<$ is indeed a relation in this specific case, it may not hold in general. Suppose $f : (\tau_1,\ \tau_2) \to \mathrm{bool}$ is a procedure with side effects on its arguments with types τ_1 and τ_2, apparently the relation is ill-defined as any side effect would interfere with the result. Therefore, the attempt of incorporating the conditionals into the type relations is extremely prone to error and pitfalls. To salvage this opportunity, the type system introduces another type of relations that are defined on terms instead of types, called "constraints".

The definition of a constraint is therefore a binary relation with the signature $(\tau_1,\ \tau_2) \to \mathrm{bool}$.

$$
\begin{aligned}
x_1 = x_2 &\Rightarrow \tau_{x_1} = \tau_{x_2} \\
x_1 \wedge x_2 &\Rightarrow x_1 \\
x_1 \wedge x_2 &\Rightarrow x_1 \\
\neg(x_1 \vee x_2) &\Rightarrow \neg x_1 \wedge \neg x_2 \\
x_1 < x_2 &\Rightarrow \langle \tau,\ x_2,\ \overline{\tau_i} \rangle \subset \langle \tau,\ x_1,\ \overline{\tau_i} \rangle \\
x_1 > x_2 &\Rightarrow \langle \tau,\ x_1,\ \overline{\tau_i} \rangle \subset \langle \tau,\ x_2,\ \overline{\tau_i} \rangle \\
x_1 < x_2 &\Rightarrow \max\left(\tau_{x_1}\right) < \min\left(\tau_{x_2}\right) \\
x_1 > x_2 &\Rightarrow \min\left(\tau_{x_1}\right) > \max\left(\tau_{x_2}\right)
\end{aligned}
$$

Figure 4. Deconstruction rules

$$
\begin{aligned}
C ::= & \ x \\
& \mid x_1 = x_2 \\
& \mid x_1 \wedge x_2 \\
& \mid x_1 \vee x_2 \\
& \mid x_1 < x_2 \\
& \mid x_1 > x_2 \\
& \mid \neg x
\end{aligned}
$$

Superficially, this seems identical to the boolean expressions defined above, and indeed they carry the same semantics. But they provide a way to formalize the concept of relation. In addition, only variables and variable-like expressions such as indexing and projection are allowed to be operated upon because functions with side-effects in general, are ill-defined mathematically and thus hard to keep track of. In this way, the typechecker can be sure that if a variable is not referred to in previous computations, its value will never change. For example, the notations in Fig. 3 such as C_t, C_f and alike are the constraints on variables that are present in expression t and f. Specifically, in the T-If type rule, for instance, constraints in $C \cup C_t$ are immediately dismissed if they contain variables that are present in V_c, meaning that their states have been altered by the side-effects in c.

The constraints are in an intermediate state between propositions and functions, in the sense that structural equality is congruent with semantic equality, and conjunction/disjunction conditions can be split further into clauses. In fact, many of these constraints have their special rules that interact with the type system to provide more typing information. However, some common inference rules are missing, namely disjunction and negation. This is because determining a boolean assignment to a disjunction clause is an NP-hard problem, and the negation rule itself does not provide much information.

In the last two rules in Fig. 4, the types of x_1 and x_2 will be narrowed down to match this constraint.

4. Algorithm

The typechecker in Erlang Core Language uses a two-pass algorithm to determine the type for every expression, as described in the paper [6]. To summarize, during the first stage, the algorithm traverses through the code and generates a set of constraints, and then solves these constraints using a set of type resolution rules. To handle situations where functions are recursive or mutually recursive *efficiently*, an extra sub-stage is introduced when generating constraints, where function calls are reduced to SCC (Strongly Connected Components) which then form a DAG (Directed Acyclic Graph). The two-pass algorithm in Erlang leverages the fact that the program is composed of interdependent functions and types of terms always satisfy previous constraints.

However, these two assumptions are counter-productive in this success type system, as a program is defined as a sequence of statements that can not be evaluated to a value, hence has no appropriate type associated, as well as that types of values can and will break previous "constraints", if there are any.

The proposed typechecker therefore does not take the similar approach, but builds a DAG of types gradually as it checks through each statement. Each vertex in the DAG can be conceptualized as a tuple of a type and its automatically generated ID, although the actual implementation is different due to performance considerations. Two nodes are connected if one is strictly less than the other, so that all types are always connected to $\bot$ with finitely many steps, and conversely, $\top$ is always connected to all types with finitely many steps.

When inferring the type of an expression, the inference algorithm will start with a given upper bound. Assume the inferred type is t of expression e after applying the typing rule T-App, then the algorithm will determine if t exists in the graph. If not, then a new vertex, t with its generated ID, will be added to the graph, as shown in 1. In addition, since some methods, referred to as procedures, can have side-effects where variables' values in the environment are altered and invalidate some constraints, these side-effects are captured and encoded as the variables that are changed, denoted as V_e. Then typechecking algorithm only proceeds after removing all constraints on v where $v \in V_e$. Additionally, T is the set of mappings from variables to type ID as a way to decouple the type objects from variables: their types can change, whereas type objects that have already been added into the graph should not be altered arbitrarily.

The actual typechecking algorithm closely resembles the typing rules mentioned above 2. When type coercion — a term's type must be updated to capture the newly implied

property — happens, the type object t in the graph only then updates to the intersection between its original type t and the inferred type that it must satisfy. A good property of this update is that, the subgraph that is rooted on t preserves previous structures, so the change is limited to a certain scope. On the other hand, types that are parametrized by t, which are also in the scope, need to be updated as they are not necessarily subtypes of t.

The aforementioned type conjunction has its counterpart, type disjunction. It closely resembles the disjunction in Erlang's success typing, that it can be only introduced by pattern matching and `if` statements, although a few points are worth mentioning after being generalized into this context. Its subtypes are not updated in order to preserve their structures, but the types that are parametrized by it are still updated in a similar manner. For example,

```
x = [1, 1]
if (b) {
    x[0] = 2
}
```

As $x[0]$ changes from $\{1\}$ to $\{2\}$, the type for x changes from $\langle\{1\},\ 2,\ \emptyset\rangle$ to $\langle[1,\ 3),\ 3,\ \{0:\{2\},\ \{1:\{1\}\}\rangle$.

To typecheck the entire program is no more different than simply checking a sequence of statements. In fact, it is indeed defined that way. Usually, if a statement passes typechecking, then its evaluated type is discarded while its environment is passed to the next statement. One exception is the `return` statement, where the following evaluation is terminated and the returned value is used to determine the type signature of caller.

Algorithm 1 The type inference algorithm

procedure INFERTYPE(e, G, T, C)
  $t,\ V_e \leftarrow$ typeof e under $G\ T\ C$
  guard $t \neq \bot$
  $G \leftarrow G \cup \{t\}$ ▷ if t is new, assign a new ID t_{ID}
  $T \leftarrow T \cup (e,\ t_{ID})$
  return $t_{ID},\ V_e$
end procedure

5. Parametrized Type and Method Overload

One limitation of this type system is that the type signatures of functions might not be informative enough to enforce meaningful type checks. This is an example from the original paper, adapted to the current syntax:

```
and(x1, x2) {
    switch (x1, x2) {
        case False, _:
            return False;
        case _, x2:
            return x2;
    }
}
```

Algorithm 2 The typechecking algorithm

procedure TYPECHECK(s, C)
  if $s =$ Assign $x\ e$ **then**
    $T(x),\ V_e \leftarrow$ inferType $e\ G\ T\ C$
    $C \leftarrow (C - C_x) \cup C[e \mapsto x]$
    return $V = \{x\} \cup V_e$
  else if $s =$ Impure e **then**
    $t,\ V_e \leftarrow$ inferType $e\ G\ T\ C$
    return V_e
  else if $s =$ Break **then**
    return $\square$
  else if $s =$ Return e **then**
    $t,\ V_e \leftarrow$ inferType $e\ G\ T\ C$
    return $t,\ V_e,\ \square$
  else if $s =$ If $c\ t\ f$ **then**
    $c_{ID},\ V_c \leftarrow$ inferType $c\ G\ T\ C$
    $C \leftarrow C - C_{V_c}$
    $C_t \leftarrow \text{toConstraints}_\text{t}\ c$
    $t_t,\ V_t \leftarrow$ checkMany $t\ C$
    $t_f,\ V_f \leftarrow$ checkMany $f\ C$
    return $t_t \cup t_f,\ V_t \cup V_f$
  else if $s =$ While $e\ ss$ **then**
    $c_{ID},\ V_c \leftarrow$ inferType $c\ G\ T\ C$
    $C \leftarrow C - C_{V_c}$
    $C_t \leftarrow \text{toConstraints}_\text{t}\ c$
    $t,\ V \leftarrow$ checkMany $t\ C$
    guard $\forall v \in V.\ T(v) \subseteq T'(v)$
  else if $s =$ Procedure $x\ ps\ ss$ **then**
    $T \leftarrow T \cup (ps, \overline{\top})$
    $t,\ V \leftarrow$ checkMany $ss\ C$
    Add $(x,\ t)$ to G and T accordingly
    return $\emptyset$
  else if $s =$ Switch $e\ \{\overline{p_i} \rightarrow \overline{ss_i}\}$ **then**
    bind the variables and infer types
    $\overline{t_i},\ \overline{V_i} \leftarrow$ checkMany $\overline{ss_i}\ C$
    return $\bigcup \overline{t_i},\ \bigcup \overline{V_i}$
  end if
end procedure

Since there always exists one branch that works for any type of x_1 or x_2, the typechecker would infer that the function signature is

$$(any,\ any) \rightarrow any$$

, the most generic type for functions of arity 2. While it indeed captures the set of all possible subtypes, the signature itself is rather unintuitive and unhelpful compared to

Algorithm 3 Typecheck a sequence of statements

procedure CHECKMANY(ss, C)
  if $ss = []$ **then**
    return $\square$
  else if $ss = s : s'$ **then**
    $t,\ V,\ \square? \leftarrow$ typeCheck s
    if $\square$ **then**
      return $t,\ V,\ \square$
    else
      $t,\ V' \leftarrow$ checkMany $ss'(C - C_V)$
      return $t,\ V \cup V'$
    end if
  end if
end procedure

statically typed languages'. One proposed solution, albeit never implemented, is to incorporate qualifiers into the type signature, i.e.

$$\forall\alpha_1\ \alpha_2.(\{false\}?(\alpha_1 \subseteq \{\{false\}\})\} \cup (\alpha_2)$$

As admitted in the original paper, this does mitigate the problem of overly generic type signatures, but it is at the cost of readability.

However, we can further generalize the idea of parametrized types as a member of the mapping between types, and all types that are parametrized by two types are constructions of the members of kind $* \to * \to *$. The construction themselves may not need to be limited to set operations or bijective mappings such as the type signature $\tau \to [\tau]$. If we treat types as if they are values in the context of higher-order type operators, we can utilize familiar concepts such as pattern matching to construct a surjective, statically determined and terminating type constructor. Consider this example:

$$f : \forall\tau.\texttt{case}\ \tau\ \texttt{of}\ \text{bool} \to \{0\};\ \tau'[]\ \to \{[]\}$$

the semantics of this example is obvious: if τ is bool, then $f(x)$ for any valid x would be the singleton type of $\{0\}$, and similarly, $f(x)$ would be of type $\{[]\}$ if τ has type $(\tau',\ n,\ \overline{\tau_i})$ for some integer n and additional typing $\overline{\tau_i}$. Note that this is not a mere change in notation compared to the qualified set expression, as the fundamental difference is the capability to evaluate the higher-order expression and obtain the normal form. Note that, however, not all expressions are valid since the value of this expression needs to be determined statically, loops and recursions, if they can exist at all, should be strictly limited so that the typechecker can reason about the termination. Furthermore, the type expression could be formally defined to have only the construction from pattern matching, as any other types of operations on types and the underlying sets would not make enough sense.

Revisiting the previous example 5, we can rewrite its type signature as follows:

$$\begin{aligned} and : \forall\beta_1,\ \beta_2.&\texttt{case}\ \beta_1 \times \beta_2\ \texttt{of} \\ &\{false\} \times \beta_2\ \to \{false\}; \\ &\{true\} \times \beta_2 \to \beta_2 \end{aligned}$$

However, there is one more question remains unanswered, that is how do we construct such a delicate type in a language where there is no way to explicitly define a type? The structure of the case expression could be thought of as an implementation of the method overloading feature prevalent in many imperative languages. For example, to produce a function signature shown above, we can have this specific function definition:

```
f(b) {
    if (b) { return b; }
    else { return b; }
}
f(l) { l[0]; return []; }
```

This style of notation can even work for method overloads with different arity. Suppose f has the following overload signatures:

$$\begin{aligned} f_1 &: (X,\ Y) \to Z_1 \\ f_2 &: A \to Z_2 \end{aligned}$$

The combined method signature f would be:

$$\begin{aligned} f : \forall\tau_1,\ \tau_2.&\texttt{case}\ \tau_1 \times \tau_2\ \texttt{of} \\ &X \times Y\ \to Z_1; \\ &A \times \emptyset \to Z_2 \end{aligned}$$

Since each method overload itself can be considered an independent function, thus type inference algorithm can be applied to the parameters to infer their types.

This feature, however, is not currently incorporated into the language due to the drastic change in type representation. In addition, although this style of pattern matching can be applied to types dependent on constant values, there has not been a suitable notation to match unions of integer ranges.

6. Conclusion

This language shows that type inference, being one of the many merits of a fully statically typed language, can be generalized into a variant of success typing such that, not only

the typing rules remain congruent and parallel to the success typing of immutable terms in a functional language, but also enables dynamic-typing-like behaviors yet still statically provable. Furthermore, the type inference algorithm even allows for totally omitting type signatures. One potential application for this type system might be parsing volatile yet structured enough markdown or configuration files, while still providing necessary type safety compare to dynamic languages.

References

[1] Alexander Aiken, Edward L Wimmers, and TK Lakshman. Soft typing with conditional types. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 163–173, 1994. 2

[2] Robert Cartwright and Mike Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 278–292, 1991. 1

[3] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. Typing local control and state using flow analysis. In *Programming Languages and Systems: 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26–April 3, 2011. Proceedings 20*, pages 256–275. Springer, 2011. 2

[4] Robert Jakob and Peter Thiemann. A falsification view of success typing. In *NASA Formal Methods: 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings 7*, pages 234–247. Springer, 2015. 1

[5] Liyi Li, Yiyun Liu, Deena Postol, Leonidas Lampropoulos, David Van Horn, and Michael Hicks. A formal model of checked c. In *2022 IEEE 35th Computer Security Foundations Symposium (CSF)*, pages 49–63. IEEE, 2022. 5

[6] Tobias Lindahl and Konstantinos Sagonas. Practical type inference based on success typings. In *Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 167–178, 2006. 1, 2, 6