Practical Type Inference Based on Success Typings

Tobias Lindahl [1]   Konstantinos Sagonas [1,2]

[1] Department of Information Technology, Uppsala University, Sweden
[2] School of Electrical and Computer Engineering, National Technical University of Athens, Greece
{tobiasl,kostis}@it.uu.se

Abstract

In languages where the compiler performs no static type checks, many programs never go wrong, but the intended use of functions and component interfaces is often undocumented or appears only in the form of comments which cannot always be trusted. This often makes program maintenance problematic. We show that it is possible to reconstruct a significant portion of the type information which is implicit in a program, automatically annotate function interfaces, and detect definite type clashes *without* fundamental changes to the philosophy of the language or imposing a type system which unnecessarily rejects perfectly reasonable programs. To do so, we introduce the notion of *success typings* of functions. Unlike most static type systems, success typings incorporate subtyping and never disallow a use of a function that will not result in a type clash during runtime. Unlike most soft typing systems that have previously been proposed, success typings allow for compositional, bottom-up type inference which appears to scale well in practice. Moreover, by taking control-flow into account and exploiting properties of the language such as its module system, success typings can be refined and become accurate and precise. We demonstrate the power and practicality of the approach by applying it to Erlang. We report on our experiences from employing the type inference algorithm, without any guidance, on programs of significant size.

Tried to make it little by little,
tried to make it bit by bit on my own...
Dressed for Success — Roxette

Categories and Subject Descriptors F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure; D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement—Documentation

General Terms Algorithms, Languages, Theory

Keywords Constraint-based type inference, success typings, subtyping, Erlang

1. Introduction

For programmers already experienced in developing programs in dynamically typed functional languages, programming is a tranquil and relatively uneventful activity, at least initially. The occasional frustrations of having to convince the type system that one really knows what she is doing are avoided. Also, since type declarations and annotations need not be typed (in), program development can progress more rapidly. Unfortunately, this freedom of expression comes with a price. Significantly less typos and other such mundane programming errors are caught by the compiler. More importantly, the freedom of *not* stating one's intentions explicitly, considerably obstructs program maintenance. In many cases, it is extremely difficult to recall or decipher how a particular piece of code — often written by some other programmer years ago — can be used. Comments are unreliable, often cryptic and confusing, and more often than not rotten. The programmer is much better off if aided by techniques and tools that can help in such situations.

Over the years, many researchers have tried to address such issues. Some have tried to impose and/or tailor a static type system to dynamically typed languages. Despite the technical depth and level of sophistication in many of the proposals, it is fair to say that so far static type systems in dynamically typed languages have enjoyed only limited success in practice. It seems that imposing a static type discipline on a language which was originally designed without one in mind is a Sisyphean task. Other researchers have taken a more low-profile approach and have built useful and successful type inference tools for different programming language paradigms. Among these, we mention soft typing systems [13] and the DrScheme [8] development environment for Scheme, the Ciao Prolog system [9], and systems developed in the context of Constraint Logic Programming languages [7] which infer types to detect errors and aid the programmer in debugging.

Unfortunately, the technology developed in the context of these systems is often very tightly coupled to the operational semantics and characteristics of the language and/or environment and as such not generally applicable. However, ideas in them provide partial inspiration for our work. In particular, from the work on soft typing systems we adopt the idea that the type system should not reject programs and from the work on (constraint) logic programming languages we get inspiration from the work on how to synthesize the call-success semantics of a program. Still, our work is done in the context of a functional language where the basic operation is pattern matching and information only flows in one direction.

More specifically, in this paper we present a method to obtain a natural typing of each function in a functional program which is both simple and powerful. Our method is simple because we address the type inference problem directly using constraint-based type inference techniques and without imposing any restrictions on function uses which are error-free in the operational semantics of the language. Our method is also powerful since it is compositional, scalable, and is able to automatically infer accurate types for recursive, higher-order, dynamically typed functional programs without requiring any type information or guidance by the user.

PPDP'06 July 10–12, 2006, Venice, Italy.
Copyright © 2006 ACM 1-59593-388-3/06/0007...$5.00.

Our contributions

- We introduce the notion of *success typings* to the functional world, a natural description of functions' behavior in a dynamically typed functional language based on pattern matching.
- We give a formalization of success typings and describe a scalable constraint-based algorithm to obtain and refine them.
- We compare success typings to typings derived by traditional static type systems.
- Using success typings we have developed an optional soft type system for Erlang that can detect definite type clashes and provide automatic documentation for large programs.

2. Our Domain and Closely Related Work

Erlang is a strict, dynamically typed functional programming language that comes with built-in support for message-passing concurrency, interprocess communication and distribution [3]. The Erlang/OTP (Open Telecom Platform) system from Ericsson is its standard implementation.

It is important to point out that although Erlang is dynamically typed, it is *type safe*. Currently, with very few exceptions where the compiler performs a rather unsophisticated function-local type inference and safely unboxes values, all values are tagged with their type during runtime. This in turn makes it possible to check the type of each value before its use as an argument to a function or a built-in operation and throw an exception if a type clash occurs. In addition to such runtime type tests, the programmer can make explicit control flow choices based on types by using pattern matching and explicit type tests in guards.

Among functional languages, Erlang is probably the one with most commercial applications. The Erlang/OTP system is actively used to write large-scale, fault-tolerant software applications mainly in the telecommunications industry. Many of these applications have been developed over long time periods, typically by large groups of programmers who do not necessarily all follow some specific style of programming. In particular, many programs are written without adhering to any (implicit) static type discipline. Nevertheless, a large portion of them never goes wrong.

Regardless of programming language and style, maintaining large applications, correcting (even simple) bugs, and understanding code written years ago by someone else is not an easy task. Previously, we have developed the Dialyzer [10], a `lint`-like tool that extracts some limited form of implicit type information from Erlang code in order to statically find obvious type clashes and report them to the user in the form of warnings. Our experiences with Dialyzer and its current uses show that it is possible to infer various forms of non-trivial type information for Erlang programs in a completely automatic and scalable way.

Inspired by the success of Dialyzer in the Erlang user community, our next goal was to design a tool that can automatically uncover the implicit type information in Erlang programs and explicitly annotate them with it. For such a tool to be successful, it is imperative that its underlying type system is tailored to the *existing* language and its current practice. Also, that the types which are inferred are understandable to programmers. This in turn implies that the type system — at least initially — has to favor simplicity over expressive power.

There has been much work in inferring type information in the context of dynamically typed languages. Next, we review the part of this work which is closely related to Erlang.

2.1 Subtyping systems and the need for subtyping in Erlang

Type systems based on subtyping try to solve sets of constraints of the form $\alpha \subseteq \beta$ (where α and β represent types) while unification based type systems in the tradition of Hindley and Milner try to solve constraints of the form $\alpha = \beta$. Subtyping systems are strictly more general than type systems built on Hindley-Milner type inference since all types expressed in Hindley-Milner type systems can be expressed in subtyping systems, though the converse is not true.

Because of the way Erlang programs are written, a type system for Erlang needs to be based on (unrestricted) subtyping. For example, consider the following Erlang function from the Erlang/OTP standard library module `pg` that manages process groups.

```
send(Pg, Mess) when is_atom(Pg) ->
  global:send(Pg, {send, self(), Mess});
send(Pg, Mess) when is_pid(Pg) ->
  Pg ! {send, self(), Mess}.
```

This is a function with two arguments, `Pg` and `Mess`, consisting of two guarded clauses. The first clause handles the case when the `Pg` argument is an atom; the second clause when `Pg` is a process identifier (an Erlang pid). When `Pg` is an atom, it denotes a globally registered process and the library function `global:send/2` is used. When it is a pid, the Erlang built-in send function (denoted `'!'`) is used to send the message (a 3-tuple).

A constructor-based type system such as Hindley-Milner can not type this function. First there needs to be some appropriate declaration that describes the type of the first argument. In this type declaration the primitive types 'atom' and 'pid' need to be wrapped in appropriate constructors which play the rôle of runtime tags. Moreover, these constructors need to be exclusive; they cannot be used in any other type. The function would then need to be rewritten to explicitly match on these constructors instead of performing type checks using guards. In short, imposing a Hindley-Milner type system on Erlang requires modifications to existing code and amounts to starting to program in a different language, not in Erlang as we currently know it. For a language with existing applications often consisting of more than one million lines of code, this is not a viable option.

In this situation, subtyping comes to the rescue. If we adopt a subtyping system that allows for disjoint union types we can simply describe the first argument as a union containing atoms and pids. This indicates that the function can be called with any subtype of this union, i.e., with any specific atom or pid. In such a scheme, the second argument can then have the type which denotes the set of all terms.

The idea of adopting a subtyping system for inferring types in dynamically typed languages is not new, not even in the context of Erlang. In 1997, Marlow and Wadler [11] proposed a subtyping system partly based on the work of Aiken and Wimmers [1]. Their approach generates a system of constraints from the code and then tries to prove that the system is solvable by reducing it until it can be showed to be consistent. Marlow and Wadler successfully applied their type system to a portion of the Erlang/OTP standard library of that time. However, to their credit, they also reported some problems with their approach. For example, pattern matching compilation causes their type system to infer wrong types for functions in certain cases where the code includes the "don't care" pattern (represented by '_'). The following implementation of the Boolean `and` function is taken from their paper [11, Section 9.3].

```
and(true, true) -> true;
and(false, _) -> false;
and(_, false) -> false.
```

Let us denote the set of all Erlang terms by $any()$. Also, let $bool()$ denote the union `true` $\cup$ `false`. Hindley-Milner type inference will derive the type $(bool(), bool()) \to bool()$ for this function. This type is correct given the definition of $bool()$ and under the constraint that these atoms are not part of any other type.

Notice however that there is nothing in this code fragment that constrains the domain of this function to $bool()$. Indeed, in Erlang the function call `and(false,42)` evaluates to `false`. In fact, no call to this function with the atom `false` in either the first or the second argument will ever raise an exception during runtime; independently of what type the value of its other argument has.

Contrary to the operational semantics of Erlang programs, the type inference algorithm of Marlow and Wadler, which allows subtyping, infers the type $\tau = (any(), \texttt{false}) \rightarrow bool()$ for this function. Looking at the code, the first clause alone provides sufficient evidence that this type is a bit counterintuitive. More importantly, the inferred type is unexpected in the sense that the type derived by Hindley-Milner is not an instance of τ. Besides oddities such as this, there are additional open issues of more practical nature in the Marlow and Wadler proposal that were never adequately addressed. For example, the type system often demands that programs are rewritten to explicitly handle failing pattern matching cases and to contain type definitions. As a result, although their proposal has significantly raised the level of type awareness among Erlang programmers, their actual type system never caught on in the Erlang community.

2.2 Soft type systems

Systems based on soft typing were first proposed by Cartwright and Fagan [6]. The aim of soft typing is to type check dynamically typed functional languages, report possible type clashes, and insert dynamic type checks at appropriate places in order to make the programs well-typed. One important property of a soft type system is that no program is ever rejected by the type checker.

Over the years, several soft type systems have been proposed for different dynamically typed languages, most notably for Scheme [13]. We also note the work of Aiken, Wimmers, and Lakshman [2] who describe a soft type system based on subtyping. The type system includes intersection, union, and conditional types. Conditional types in particular, are introduced to reflect the fact that in case expressions certain clauses can be unreachable and should not contribute to the type of the case expression. In this way, control flow *can* affect the inferred types. The type system of Aiken et al. chooses accuracy over readability; the inferred types end up being complex type expressions which include constraints. Although in our work we do adopt the idea that case clauses which are unreachable should not influence the type of case expressions, we aim for readability and simplicity in the types we infer, instead of maximum expressive power. As mentioned, our goal is to infer typings that describe the behavior of functions in a way that is intuitively clear to Erlang programmers.

In the context of Erlang, a proposal for a soft type system has been made by Nyström [12]. Its type inference algorithm is based on a dataflow analysis which is guided by optional user annotations. The main idea of the type system is that the user should supply annotations at all interface points. Then the dataflow analysis will report inconsistencies in these annotations and will warn about all program points where type clashes can possibly occur. Because of the inherently dynamically-typed programming style practiced by many Erlang programmers, the warnings are plentiful, especially if no type annotations are provided.

We take the completely opposite approach. To eliminate noise and all false warnings, we optimistically assume that any expression will evaluate successfully if we cannot prove that it will result in some type clash. Besides differences in the philosophy of the approach, there are also technical differences. For example, since Nyström's dataflow analysis only propagates information forward in the control flow, the only type information that it can infer at function entry points is information about how the function is currently used. This is useful information to derive, but it relies on the type system having complete knowledge about a function's intended usage. Thus, the method is not modular. Moreover, Nyström's system cannot be used for automatically providing documentation for library modules, which is one of our goals.

2.3 Our Goals

Our main goal is to make uncover the implicit type information in Erlang code and make it explicitly available in programs. Because of the sizes of typical Erlang applications, the type inference should be *completely automatic* and faithfully respect the operational semantics of the language. Moreover, it should impose *no code rewrites* of any kind. The reason for this is simple. Rewriting, often safety critical, applications consisting of hundreds of thousand lines of code just to satisfy a type inferencer is not an option which will enjoy much success. However, large software applications have to be maintained, and often not by their original authors. By automatically revealing the type information that is already present, we provide automatic documentation that can evolve together with the program and will not rot. We also think that it is important to achieve a balance between precision and readability. Last but not least, the inferred typings should *never be wrong*.

During development, the type information can be used to verify the intentions of programmers and help them discover bugs at an early stage. The type inference should be able to infer reasonable typings even when some part of the code is *not available*, for example if some parts have not been written yet. Of course, when the code becomes available the type information can be taken into account and the inferred typings can become more precise.

In order to meet these goals, we build our type inference around the notion of *success typings* that we define in Section 4.

3. Language and Types

3.1 Programming language

A compact description of a mini-Erlang programming language is shown in Figure 1(a). For simplicity we only deal with a subset of Core Erlang [4] here. However, it is easy to extend our analysis to handle all of Core Erlang.[1] The Core Erlang language constructs that we omit are the `try-catch` and `receive` expressions, which can be handled as relatively minor variations of `case` expressions, and sequence operators, which can be treated as `let` expressions where the variable is never used.

Most of the language of Figure 1(a) is fairly standard, but we comment on some issues. Evaluation is strict. Functions are not curried but explicitly take zero or more input arguments. The language is higher order: functions can be used as arguments and returned as results. Pattern matching is generalized to `case` statements. A term t matches a pattern p if the variables in p can be bound so that p represents a term syntactically identical to t. A clause is chosen if the patterns in its head match and the constraints in the guard are satisfied. Variables in patterns of a clause head are fresh, but by the use of equality constraints in clause guards they can be made to refer to bound variables. If a clause does not contain any explicit guards, its Core Erlang translation has `true` as its guard. Guards are restricted to be conjunctions of simple constraints on variables and constants. Core Erlang allows for some other guard constructs, but the type information in these can be expressed in this slightly more restricted form.

We show an example of Erlang to mini-Erlang translation. The function `and` from Section 2.1 has the following translation:

[1] Core Erlang is the intermediate language that all Erlang programs are translated to by the Erlang/OTP compiler.

$$\begin{array}{lcl}
e & ::= & X \mid c(e_i,\ldots,e_n) \mid e_1(e_2,\ldots,e_n) \mid f \mid \\
 & & \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid \\
 & & \texttt{letrec}\ x_1 = f_1,\ldots,x_n = f_n\ \texttt{in}\ e \mid \\
 & & \texttt{case}\ e\ \texttt{of}\ (p_1 \to b_1);\ldots;\ (p_n \to b_n)\ \texttt{end} \\
f & ::= & \texttt{fun}(x_1,\ldots,x_n) \to e \\
p & ::= & p'\ \texttt{when}\ g \\
p' & ::= & x \mid c(p'_1,\ldots,p'_n) \\
g & ::= & g_1\ \texttt{and}\ g_2 \mid x_1 = x_2 \mid \texttt{true} \mid \texttt{is\_atom}(x) \mid \\
 & & \texttt{is\_integer}(x) \mid \ldots
\end{array}$$

(a) The mini-Erlang programming language

$$\begin{array}{lcl}
T & ::= & \mathit{none}() \mid \mathit{any}() \mid V \mid c(T_1,\ldots,T_n) \mid \\
 & & (T_1,\ldots,T_n) \to T' \mid T_1 \cup T_2 \mid T\ \mathit{when}\ C \mid P \\
V & ::= & \alpha, \beta, \tau \\
P & ::= & \mathit{integer}() \mid \mathit{float}() \mid \mathit{atom}() \mid \mathit{pid}() \mid 42 \mid \texttt{foo} \mid \ldots \\
C & ::= & (T_1 \subseteq T_2) \mid (C_1 \wedge \ldots \wedge C_n) \mid (C_1 \vee \ldots \vee C_n)
\end{array}$$

(b) Type expressions

Figure 1. A description of the programming language and type expressions.

```
let And = fun(X, Y) ->
            case <X, Y> of
              <true, true> when true -> true;
              <false, _> when true -> false;
              <_, false> when true -> false;
            end
```

where the `<...>` denotes the product constructor that exists in Core Erlang but not in Erlang.

3.2 Types

Types represent sets of values and are denoted by type expressions. If a value v is in the set represented by a type τ, we say that the value has this type and write $v \in \tau$. Subtyping is expressed as set inclusion and denoted as $\tau_1 \subseteq \tau_2$.

Figure 1(b) shows the syntax of type expressions. Following the notation of the `edoc`[2] tool, types are written with parentheses to distinguish them from Erlang atoms. Primitive types are the expected ones in Erlang, such as $\mathit{integer}()$, $\mathit{float}()$, $\mathit{atom}()$, and $\mathit{pid}()$ (denoting process identifiers). There is a largest type, $\mathit{any}()$, representing all values and a smallest type, $\mathit{none}()$, representing the empty set. Type variables are represented by the Greek letters α, β and τ. We use the notation $\bar{\alpha}$ as a shorthand for $\alpha_1, \ldots, \alpha_n$ and write $\bar{v} \in \bar{\alpha}$ whenever $v_i \in \alpha_i, 1 \leq i \leq n$. Structured types, denoted $c(T_1, \ldots, T_n)$ in the figure, are tuples and cons cells. To simplify the handling of pattern matching, we also consider products as structured terms. Also, to gain better precision in our analysis, we allow for singleton types such as the integer 42 or the atom `foo`. This does not cause any extra complexity since our type system is based on subtyping anyway.

Union types are expressed with the $\cup$ symbol. We allow for any disjoint union, including unions of singleton types such as $1 \cup 2$. Since these unions can become large or even infinite, in our analysis we impose a fixed size limit after which the union is widened to a supertype. For example, if the union limit is three the union type $1 \cup 2 \cup 3 \cup 4$ will be widened to $\mathit{integer}()$.

Since functions are not curried, function types explicitly show the number of arguments. For example, a function with two arguments is represented by $(T_1, T_2) \to T_3$. During type inference it is useful to bind constraints to a function type and achieve a variant of bounded quantification. For this, we use constrained types of the form $T\ \mathit{when}\ C$, where C is a set consisting of nested conjunctions ($\wedge$) and disjunctions ($\vee$) of subtype constraints. Constrained types should be interpreted as:

$$T\ \mathit{when}\ C ::= \begin{cases} \mathit{Sol}(T) & \text{if } \mathit{Sol} \text{ is a solution to } C \\ \mathit{none}() & \text{if } C \text{ has no solution} \end{cases}$$

Sol is a mapping from type expressions and type variables to concrete types. Concrete types include all type expressions with the exception of constrained types and type variables. Our constrained types are similar to conditional types [2] but where we condition the type over a constraint set, conditional types use intersection types as in the following:

$$\alpha?(\beta \cap \tau) = \begin{cases} \alpha & \mathit{when}(\beta \cap \tau) \neq \emptyset \\ \mathit{none}() & \mathit{when}(\beta \cap \tau) = \emptyset \end{cases}$$

While the ability to handle conditional types constitutes a possible extension of our work, we do not include such types. In the work of Aiken and Wimmers [2] conditional types are used to capture the control flow in case statements, but we use a different approach. This is described in more detail in Section 5.

4. Success Typings

4.1 Basic idea

Assume that a function f is described by the type signature $(\bar{\alpha}) \to \beta$ in some system of types. In a statically typed language, the standard interpretation of this signature is that *provided* that $\bar{p} \in \bar{\alpha}$, the function application $f(\bar{p})$ can evaluate to a value $v \in \beta$ without type errors (which are expressible in this type system). In other words, $\bar{\alpha}$ is the largest type (if the type signature is a *principal type*) for which the type system can prove type safety without dynamic type tests. Because of the requirement to *prove* type safety statically, sometimes the domain of the function is unnecessarily restricted to a smaller set of values than the function can accept and evaluate without type errors during runtime. Also, note that β depends on $\bar{\alpha}$, so if $\bar{\alpha}$ has been constrained, β expresses the restricted range of the function under the restricted type domain.

Unlike statically typed languages, we are not concerned with proving type safety — this is already provided by the underlying implementation. Also, we will not try to use the inferred types for removing dynamic type tests — at least not in this work. We are instead interested in capturing the biggest set of terms for which we can be sure that type clashes will definitely occur. Instead of keeping track of this set, we will design an algorithm that infers its complement, a function's *success typing*. A success typing is a type signature that over-approximates the set of types for which the function can evaluate to a value. The domain of the signature includes all possible values that the function could accept as parameters, and its range includes all possible return values for this domain.

DEFINITION 1 (Success Typings). *A success typing of a function f is a type signature, $(\bar{\alpha}) \to \beta$, such that whenever an application $f(\bar{p})$ reduces to a value v, then $v \in \beta$ and $\bar{p} \in \bar{\alpha}$.*

Note that there is a fundamental difference between success typings and type signatures of a static type system. The difference is that success typings capture all possible intended uses of a function and then some. In particular, success typings capture some uses that might result in a type clash and some type-correct uses which never evaluate to a value (either due to non-termination or

[2] A documentation tool for Erlang in the spirit of `javadoc`.

because of throwing an exception). However weak this might seem to aficionados of static typing, success typings have the property that they capture the fact that if the function is used in a way not allowed by its success typing (e.g., by applying the function with parameters $\bar{p} \notin \bar{\alpha}$) this application will *definitely* fail. This is precisely the property that a defect detection tool which never "cries wolf" needs. Also, success typings can be used for automatic program documentation because they will never fail to capture some possible — no matter how unintended — use of a function.

4.2 Examples

Let us revisit the Boolean `and` function we saw in Section 2.1.

```
and(true, true) -> true;
and(false, _) -> false;
and(_, false) -> false.
```

Assume that $\mathit{bool}() = \texttt{true} \cup \texttt{false}$.

A static type system based on Hindley-Milner constructor-based type inference has no choice but to derive the typing

$$(\mathit{bool}(), \mathit{bool}()) \rightarrow \mathit{bool}().$$

Notice that, at least in the eyes of programmers used to a dynamically typed language, this typing unnecessarily restricts the domain of this function. Indeed, there is nothing that can possibly "go wrong" here if the function is called as e.g. `and(42,false)`. Since some programmers value freedom of expression more than obtaining type safety guarantees at compile time, it might be extremely difficult to convince them to adopt such a system.

A static type system based on subtyping, such as the one by Marlow and Wadler, might derive the typing

$$(\mathit{any}(), \texttt{false}) \rightarrow \mathit{bool}()$$

for this function. Notice that although counter-intuitive, this typing is correct from a static type system's point of view. This typing allows complete freedom in the value of the first argument provided that the second argument is the value `false`. Once again, what's happening here is that callers of this function are constrained to a smaller domain than what the function is prepared to accept. This is a general phenomenon. In some way or another, all static type systems are *pessimistic*.

Success typings aim to avoid such situations. To do so, they adopt an *optimistic* attitude and approximate types in the opposite direction. For example, a success typing for this function is

$$(\mathit{any}(), \mathit{any}()) \rightarrow \mathit{any}()$$

which trivially satisfies the condition of Definition 1. Naturally, we are interested in inferring success typings with more type information than what the typing shown above contains. Indeed, the success typing that the algorithm we present in the next section will infer is

$$(\mathit{any}(), \mathit{any}()) \rightarrow \mathit{bool}().$$

It is easy to see that this typing describes all intended uses of the `and` function.

In the example above, the success typing we infer is quite weak in the sense that it will not catch any type error in calls to the `and` function. For example, it cannot capture that the call `and(42,gazonk)` will throw a runtime exception. Notice however that it *will* catch type clashes in matching a value other than `true` or `false` against the result of this function. As we will see next, there are many situations where the inferred success typings are detailed and precisely capture the intended uses of functions. For example, the success typing inferred for the following function

```
add1(X) when is_integer(X) -> X + 1.
```

is $(\mathit{integer}()) \rightarrow \mathit{integer}()$. Note that to derive this typing the built-in function for addition, which in Erlang is overloaded and handles floats as well as integers, needs to be instantiated for integers. Our type inferencer has hard-coded knowledge about all Erlang built-ins, represented in a restricted form of dependent types. Finally, for the following function

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

our type inferencing algorithm will detect a type violation, which is expressed by assigning a typing such as $(\mathit{any}()) \rightarrow \mathit{none}()$ to it. In such cases, we say that no success typing can be inferred for the function.

A final note Note that since the type signature $(\overline{\mathit{any}()}) \rightarrow \mathit{any}()$ *is* a success typing, the analysis is free to use this signature for all functions which are unknown; because e.g. their code is not available. Besides making the analysis modular, it allows for some type clashes to be discovered early in the development process, even during rapid prototyping and random experimentation. The practical benefits of this property should not be dismissed or underestimated.

5. Inferring Success Typings

The algorithm for inferring success typings has two phases. In the first, the code is traversed and constraints are generated using derivation rules. In the second, we try to find a solution to the constraints and this solution constitutes the success typing. We describe the process in more detail by explaining constraint generation (Section 5.1), constraint solving (Section 5.2), and finally the algorithm that ties them together (Section 5.3).

5.1 Constraint generation

Figure 2 shows the rules for constraint generation. In the rules, A represents an environment with bindings of variables of the form $\{\ldots, x \mapsto \tau_x, \ldots\}$ and C represents nested conjunctions and disjunctions of subtype constraints in the same form as in the constrained types in Section 3.2:

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

We will use equality constraints, $T_1 = T_2$, as shorthands for $(T_1 \subseteq T_2) \wedge (T_2 \subseteq T_1)$. The judgment $A \vdash e : \tau,\ C$ should be read as "given the environment A the expression e has type $\mathit{Sol}(\tau)$ whenever Sol is a solution to the constraints in C".

The VAR, STRUCT and LET rules are standard. Constants can be typed by the STRUCT rule by viewing primitive types as nullary constructors. The ABS rule binds the constraints from the function body to its type, but exports no constraints. In this way the type of a function can be influenced by outer constraints on the free variables, but the constraints from the function body cannot affect the types of the free variables outside the function body.

A `letrec` statement binds a number of function declarations to *recursion variables*. The scope of the recursion variables includes both the function declarations and the body of the `letrec` statement. The LETREC rule assigns fresh type variables to the recursion variables and then adds equality constraints on the function types and the types of the recursion variables.

The PAT rule slightly abuses notation. As described in Section 3.1 the guards in a pattern can be expressed as a conjunction of simple type constraints on variables such as $\texttt{is\_integer}(x)$, $\texttt{is\_atom}(x)$, etc. and by using equality constraints on variables. The translation of these into constraints on types and type variables is straightforward and omitted for brevity. The rule states that the guard must evaluate to `true` under the translated constraints which is equivalent to stating that the constraints must have a solution.

In `case` expressions, it is enough that one clause can be taken in order for the whole expression to have a success typing. This is captured by introducing a disjunction of constraints in the CASE

$$\frac{}{A \cup \{x \mapsto \tau\} \vdash x : \tau,\ \emptyset} \qquad \text{[Var]}$$

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

$$\frac{A \vdash e_1 : \tau_1, C_1 \quad A \cup \{x \mapsto \tau_1\} \vdash e : \tau_2,\ C_2}{A \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2,\ C_1 \wedge C_2} \qquad \text{[Let]}$$

$$\frac{A \cup \{x_i \mapsto \tau_i\} \vdash f_1 : \tau_1',\ C_1\ \ldots\ f_n : \tau_n',\ C_n \quad e : \tau,\ C}{A \vdash \texttt{letrec}\ x_1 = f_1, \ldots, x_n = f_n\ \texttt{in}\ e : \tau, C_1 \wedge \ldots C_n \wedge C \wedge (\tau_1' = \tau_1) \wedge \ldots \wedge (\tau_n' = \tau_n)} \qquad \text{[Letrec]}$$

$$\frac{A \cup \{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\} \vdash e : \tau_e, C}{A \vdash \texttt{fun}(x_1, \ldots, x_n) \to e : \tau,\ (\tau = ((\tau_1, \ldots, \tau_n) \to \tau_e\ \mathit{when}\, C))} \qquad \text{[Abs]}$$

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

$$\frac{A \vdash p : \tau,\ C_p \quad A \vdash g : \texttt{true},\ C_g}{A \vdash p\ \texttt{when}\ g : \tau,\ C_p \wedge C_g} \qquad \text{[Pat]}$$

$$\frac{\begin{array}{c} A \cup \{v \mapsto \tau_v | v \in Var(p_1)\} \vdash p_1 : \alpha_1,\ C_1^p,\ \ b_1 : \beta_1,\ C_1^b \\ \vdots \\ A \vdash e : \tau,\ C_e \quad A \cup \{v \mapsto \tau_v | v \in Var(p_n)\} \vdash p_n : \alpha_n,\ C_n^p,\ \ b_n : \beta_n,\ C_n^b \end{array}}{\begin{array}{r} A \vdash \texttt{case}\ e\ \texttt{of}\ p_1 \to b_1; \ldots p_n \to b_n\ \texttt{end} : \beta,\ \ C_e \wedge (C_1 \vee \ldots \vee C_n) \\ \text{where } C_i = ((\beta = \beta_i) \wedge (\tau_i = \alpha_i) \wedge C_i^p \wedge C_i^b) \end{array}} \qquad \text{[Case]}$$

Figure 2. Derivation rules

rule. Each disjunct contains the constraints that need to be satisfied for the corresponding clause to contribute to the success typing. Intuitively, if a clause is taken at runtime, the type of each incoming argument and the corresponding pattern must be equal, and the constraints from the clause guard must be satisfied. The type of the whole `case` expression equals the type of the clause body. Note that by introducing a disjunction we separate the constraints from different clauses. At the time we solve the constraints we can choose how to interpret the disjunctions, and thus choose the level of abstraction at `case` expressions. We elaborate more on this point in Section 5.2.

Finally, note that our App rule is quite unorthodox. In traditional subtyping systems the type of an application is downwards bounded by the type of the function's range. This ensures that all possible return values are handled, possibly by inserting narrowers to make it a smaller type. As mentioned, we are not concerned with type safety, but with avoiding false alarms. We therefore let the range of the function type constitute an upper bound of the type of the application in order to avoid unnecessarily over-approximations. Intuitively, if an application succeeds the returned value must be a subtype of the range of the function type. If the type of an application is later constrained to be a smaller type, we can optimistically assume that this is true since it is necessary for evaluation and also possible based on the available information about the function.

5.2 Constraint solving

Disjunctions can only be introduced by the Case rule, where each disjunct corresponds to a clause in the `case` expression. A disjunctive normal form of constraints is a constraint set consisting of a top-level disjunction where all the parts consist of conjunctions. If the constraints would be transformed into disjunctive normal form, each conjunction would correspond to a program trace. Such a transformation would cause the number of constraints to explode in the presence of several `case` expressions. To avoid this explosion, we keep the constraints in the generated form, keeping the one to one correspondence between disjunctions and case clauses.

Let Sol be a mapping from type expressions and type variables to concrete types. Concrete types include all type expressions with the exception of constraints and type variables. We say that Sol is a solution to a constraint set C, and write $Sol \models C$, if:

$$\begin{array}{lcl} Sol \models T_1 \subseteq T_2 & \iff & none() \subset Sol(T_1) \subseteq Sol(T_2) \\ Sol \models C_1 \wedge C_2 & \iff & Sol \models C_1, Sol \models C_2 \\ Sol \models C_1 \vee C_2 & \iff & \left\{ \begin{array}{l} Sol_1 \models C_1, Sol_2 \models C_2, \\ Sol = Sol_1 \sqcup Sol_2 \end{array} \right. \end{array}$$

where $Sol_1 \sqcup Sol_2$ denotes the point-wise least upper bound of the solutions. In words: a solution satisfies a subtype constraint if the mapping satisfies the subtype constraint and neither of its constituents is $none()$. A solution of a conjunction of constraints must satisfy all conjunctive parts and a solution to a disjunction of constraints is the point-wise least upper bound of the solutions of all disjuncts. Furthermore, if a constraint set has no solution it can be assigned the solution $\perp$ which represents a solution that maps all type expressions to $none()$. Note that $\perp \sqcup Sol = Sol$. So, as long as the set of constraints from one clause in a `case` expression has a solution other than $\perp$, the constraints from the whole `case` expression also have a solution other than $\perp$.

$$
\begin{array}{lcl}
\mathtt{solve}(\bot, \_) & = & \bot \\
\mathtt{solve}(Sol, \alpha \subseteq \beta) & = & \begin{cases} Sol & \textit{when } Sol(\alpha) \subseteq Sol(\beta) \\ Sol[\alpha \mapsto T] & \textit{when } T = Sol(\alpha) \sqcap Sol(\beta) \neq none() \\ \bot & \textit{when } T = Sol(\alpha) \sqcap Sol(\beta) = none() \end{cases} \\
\mathtt{solve}(Sol, Conj) & = & \begin{cases} Sol & \textit{when } \mathtt{solve\_conj}(\mathtt{solve}(Sol, Conj)) = Sol \\ \mathtt{solve}(Sol', Conj) & \textit{when } Sol' = \mathtt{solve\_conj}(\mathtt{solve}(Sol, Conj)) \neq Sol \end{cases} \\
\mathtt{solve}(Sol, Disj) & = & \begin{cases} \bigsqcup Sol' & \textit{when } Sol' \neq \emptyset \\ \bot & \textit{when } Sol' = \emptyset \end{cases} \textit{where} \begin{cases} Sol' = \{S \mid S \in PS, S \neq \bot\} \\ PS \ = \{\mathtt{solve}(Sol, C) \mid C \in Disj\} \end{cases} \\
\mathtt{solve\_conj}(\bot, \_) & = & \bot \\
\mathtt{solve\_conj}(Sol, C_1 \wedge \ldots \wedge C_n) & = & \mathtt{solve\_conj}(\mathtt{solve}(Sol, C_1), C_2 \wedge \ldots \wedge C_n) \\
\mathtt{solve\_conj}(Sol, C) & = & \mathtt{solve}(Sol, C)
\end{array}
$$

Figure 3. Algorithm for solving constraints

5.3 Algorithm

We have described the two phases of the inference algorithm, but there are some issues that need to be described in more detail. While applying the derivation rules of Figure 2 we store some additional information. For example, when applying the ABS rule, we store the constraints corresponding to the function for easy access. Similarly, in `letrec` expressions, the binding between recursion variables and function types is recorded so that recursive functions receive a special treatment.

For efficiency reasons, the analysis first constructs the global function call graph, which describes the dependencies between functions. The call graph is a directed graph with functions as nodes and an edge (f, g) whenever f calls g. Mutually dependent functions form cycles, and the call graph is condensed to its strongly connected components (SCCs). In this way, we end up with a directed acyclic graph (DAG). This DAG is sorted topologically and the analysis infers success typings for the functions by analyzing its nodes (i.e., the SCCs of the function call graph) in a bottom-up fashion.

The constraint solver is written in Erlang. Type constraints are generated and solved at the granularity of a single function according to the algorithm in Figure 3. The basic idea is to iteratively solve all constraints in a conjunction until either a fixpoint is reached or the algorithm encounters some type clash and fails by assigning the type $none()$ to a type expression. The starting point for Sol is a mapping where all type expressions are mapped to $any()$, with the exception for the types of all recursion variables that are mapped to $none()$. The following example shows how this gives us a possibility to handle self-recursive calls.

Consider the following implementation of a function that removes all elements in odd-numbered positions from a list.

```
letrec DropOdd =
  fun(L) ->
    case L of
      [] when true -> [];
      [_] when true -> [];
      [_,H|T] when true -> [H|DropOdd(T)];
    end
in
...
```

Assume that the type of the recursion variable `DropOdd` has the type τ_D, and the other type variables are subscripted with their original variable names. We will not give the complete derivation of the constraints, but we will concentrate on the recursive application in the third clause.

$$
\frac{\vdots}{A \vdash \mathtt{DropOdd(T)} : \beta, C_1 = ((\tau_D = (\alpha_1) \rightarrow \alpha) \wedge (\beta \subseteq \alpha) \wedge (\tau_T \subseteq \alpha_1))} \ [\textrm{APP}]
$$

The function body is a `case` expression, so by the CASE rule it will yield a disjunction of the constraints from the three clauses. In the first iteration of the constraint solving algorithm the type of the recursion variable `DropOdd` is $\tau_D \mapsto none()$. This will make the constraint $\tau_D = (\alpha_1) \rightarrow \alpha$ fail. Let the solutions for the ith disjunction be denoted by Sol_i, the solution for the whole case statement be Sol, and the output of the case statement be τ_c. We then have

$$
\begin{aligned}
Sol_1 &= \{\tau_L \mapsto \mathtt{[]}, \tau_c \mapsto \mathtt{[]}\} \\
Sol_1 &= \{\tau_L \mapsto list(), \tau_c \mapsto \mathtt{[]}\} \\
Sol_3 &= \bot \\
\Longrightarrow \ Sol &= Sol_1 \sqcup Sol_2 \sqcup Sol_3 \\
&= \{\tau_L \mapsto list(), \tau_c \mapsto \mathtt{[]}\}
\end{aligned}
$$

where we used $list()$ as a shorthand for the type $list(any())$. [3] The recursion variable then gets assigned the current type of the function.

$$\tau_D \mapsto (\tau_L) \rightarrow \tau_c = (list()) \rightarrow \mathtt{[]}$$

The solution algorithm iterates once again since the type assigned to τ_D has changed. This time the constraints from the recursive application in the third clause have a solution.

$$
\left\{ \begin{array}{l} \tau_D \mapsto (list()) \rightarrow \mathtt{[]}, \alpha_1 \mapsto list(), \\ \alpha \mapsto \mathtt{[]}, \beta \mapsto \mathtt{[]}, \tau_T \mapsto list() \end{array} \right\} \models \left(\begin{array}{l} \tau_D = (\alpha_1) \rightarrow \alpha \\ \wedge(\beta \subseteq \alpha) \\ \wedge(\tau_T \subseteq \alpha_1) \end{array} \right)
$$

so the solution of the constraint disjunction is

$$
\begin{aligned}
Sol_1 &= \{\tau_L \mapsto \mathtt{[]}, \tau_c \mapsto \mathtt{[]}\} \\
Sol_1 &= \{\tau_L \mapsto list(), \tau_c \mapsto \mathtt{[]}\} \\
Sol_3 &= \{\tau_L \mapsto list(), \tau_c \mapsto list()\} \\
\Longrightarrow \ Sol &= \{\tau_L \mapsto list(), \tau_c \mapsto list()\}
\end{aligned}
$$

[3] The list type is the only recursive type in the language and is parameterized by its contents such that $list(T) = cons(T, list(T)) \cup nil$. However, the parameter T is some concrete type. For example, if nothing is known about the contents of some list, this list is represented as $list(any())$. If it is subsequently determined that the elements of this list must be subtypes of integers, this list is represented as $list(integer())$.

and the new type of the function becomes

$$\tau_D \mapsto (list()) \rightarrow list()$$

which makes the solution algorithm reach a fixpoint, so we have found the success typing for the function. The iterative way of solving constraints for self-recursive functions easily extends to SCCs of mutually dependent functions by iterating over the functions of each SCC until a fixpoint is reached.

Note that in the example above, we reach a fixpoint since the recursive type $list()$ is collapsed at the recursive call. The reason why we treat lists in a special way is that it is the by far most common recursive type in Erlang. However, other recursive types such as for example trees are also common. The typical way to build these structures in Erlang is to use nested tuples. Consider the function `tree_to_list` which transforms a binary tree to a list using inorder traversal.

```
tree_to_list(nil) ->
  [];
tree_to_list({Left, Data, Right}) ->
  tree_to_list(Left) ++ [Data|tree_to_list(Right)].
```

The type of such a tree could succinctly be expressed in either of the following ways

$$\texttt{-type}\ tree() = \texttt{nil} \cup \{tree(), any(), tree()\}.$$
$$\texttt{-type}\ tree(X) = \texttt{nil} \cup \{tree(X), X, tree(X)\}.$$

but since currently there is no mechanism to declare user-defined types, recursive or otherwise, uses of such recursive data types cannot be recognized. With the algorithm we have described, during type inference, constraint solving would expand this type indefinitely. To ensure termination we use depth-k abstraction. When the depth of a compound term grows larger than a limit k we abstract that subterm to the type $any()$. This, together with the union limit described in Section 3.2, gives us a way to limit the size of types so that they cannot grow indefinitely. [4]

DEFINITION 2. *A solution Sol is more general than a solution Sol' iff for some type variable τ we have $Sol'(\tau) \subset Sol(\tau)$.*

Note that the solution for failing constraints $\perp$ is not more general than any other solution since all variables are mapped to $none()$.

PROPOSITION 1 (Monotonicity). *In all steps in the algorithm, the output solution, if any, cannot be more general than the provided input solution.*

PROOF. Note that we can view the constraints as a tree where the leaves are simple subtype constraints and the inner nodes are either conjunctions or disjunctions. Assume that the output solution from a child node cannot become more general than the input solution to the child node. At the leaves, the right hand side of the subtype constraint does not change, and the left hand side can only become more specific. Thus the assumption holds at the base case.

For the inner nodes, we have two cases:

1. The solution of a conjunction is a fixpoint of all partial solutions. By the assumption, the partial solutions cannot be more general than the input solution. So if a fixpoint is reached, it cannot be more general than the input solution.
2. For disjunctions the output solution is the point-wise least upper bound of all partial solution. By the assumption, the partial solutions cannot be more general than the input solution, so neither can the least upper bound.

[4] The depth of the recursive $list$ type is defined based on the content of the list rather than on its length. For example, the type $list(bool())$ has depth 2.

By induction on the structure of the constraints no output solution can be more general than the input solution. □

PROPOSITION 2 (Termination). *Given an initial solution where all variables are mapped to $any()$, the algorithm terminates and produces a solution to the set of constraints.*

PROOF. The only place the algorithm loops is when faced with a conjunction of constraints, where it loops until a fixpoint is reached. By Proposition 1 none of the partial solutions in the conjunction can be more general than the input solution. The k-depth abstraction guarantees that a solution cannot become more precise indefinitely. Thus, a fixpoint must eventually be reached, and the algorithm terminates. □

5.4 Some examples of inferred success typings

We show examples of success typings on some Erlang code. We do so, to discuss pros and cons of the type expressions we currently employ. In all examples, we show the success typing as an Erlang comment directly above the function's code. With the exception of using `|` rather than $\cup$, this is precisely how our analysis annotates programs. First, let us consider the function

```
%% (integer() ∪ list()) → integer() ∪ atom()
foo(X) when is_integer(X) -> X + 1.
foo(X) -> list_to_atom(X).
```

Its success typing tells us that `foo` can be called with both integers and lists and will return either an integer or an atom. Note that the function will indeed fail if called with anything outside the stated domain. This is obvious in the first clause since it is guarded with an explicit type test. In the second clause, the call to the built-in function `list_to_atom` will fail if its argument is not a list.

Note that the success typings we currently maintain, such as the above, do not keep track of dependencies between the input and the output type. In other words, at call sites of `foo` we cannot say that the input type $integer()$ will result in an $integer()$ as output. This could be captured by conditional types and intersection types [2]. A typing of this function would look something along the lines of

$$\begin{gathered}\forall\alpha.(\alpha) \rightarrow (integer()?(\alpha \cap integer())) \\ \cup\ (atom()?(\alpha \cap list())) \\ \text{where } \{\alpha \subseteq integer() \cup list()\}\end{gathered}$$

This is undeniably a more descriptive type since it expresses the correspondence between input and output types, but it is also considerably less readable. In short, we currently sacrifice expressiveness for simplicity and readability.

The next example includes a function call to the function `add1` which was defined in Section 4.2. Recall that `add1` has the success typing $(integer()) \rightarrow integer()$.

```
%% (integer()) → ok1
bar(X) ->
  case add1(X) of
    42 -> ok1;
    gazonk -> ok2
  end.
```

The success typing of this function reflects that the second case clause can never match since the range of the function `add1` only includes integers. Finally, consider the following function which uses the function `foo` given above.

```
%% (integer()) → ok1 ∪ ok2
baz(X) when is_integer(X) ->
  case foo(X) of
    42 -> ok1;
    gazonk -> ok2
  end.
```

A type signature for `foo` that kept track of input-output type dependencies would make it possible to detect that the second clause is unreachable. Note that even though the type signature we infer is an over-approximation, it is a correct success typing.

6. Practical Aspects of Success Typings

Our definition of success typings is a solid framework for describing succinctly the most general way that functions can be used. This description is 'most general' in the sense that it allows us to reason about open programs, i.e., programs for which we do not have complete information about all calls to their functions. We have already argued why, in the context of a dynamically typed language such as Erlang, this is the appropriate thing to do from a practical standpoint. However, sometimes the success typings are so general that a function's *intended* use is lost in abstraction. For example, consider the following function which naïvely calculates the length of a list.

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

The success typing of this function, shown above its definition, captures the intention of the programmer quite precisely. However, if we decide to do a simple program transformation and make this function tail-recursive, which is a common practice in functional languages, we end up with the following two functions.

```
%% (list()) → any()
length_2(List) -> length_3(List, 0).

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

At first, it might seem surprising that the return type of `length_3` is now $any()$ rather than $integer()$. One might even jump to the conclusion that success typings are unnecessarily general and, as such, quite useless. However, notice that this success typing succinctly captures all possible applications of `length_3` which will not result in a type error. Among them is the call `length_3([a,b,c], 3.14)` which will return `6.14`, and the call `length_3([], gazonk)` which will return `gazonk`. One might argue this is not what the programmer had in mind when the function was written, but as explained before our intention is to never try to outsmart the programmer. Still, we also find the situation sub-optimal and we will improve on it as explained below.

6.1 Refined success typings

Assume that the two calls to `length_3` in the example above are the only calls to this function. It is then easy to see that, since the self-recursive loop is started with the integer `0` in the second argument, the second argument will be a subtype of $integer()$ in all subsequent self-recursive calls. Since the loop can terminate only by entering the leaf clause, the type of `N` *must* be $integer()$ for the function to return a value. Also, the return type of the function cannot be anything but $integer()$. By reasoning about the input types of `length_3` this way, we can say something more refined about the programmer's intentions. In order to capture this line of reasoning, we introduce the notion of *refined success typings*.

DEFINITION 3 (Refined Success Typings). *Let f be a function with success typing $(\bar{\alpha}) \to \beta$. A refined success typing for f is a typing of the form $(\bar{\alpha}') \to \beta'$ such that*

1. *$\bar{\alpha}' \subseteq \bar{\alpha}$ and $\beta' \subseteq \beta$, and*
2. *for all $\bar{p} \in \bar{\alpha}'$ for which the application $f(\bar{p})$ reduces to a value, $f(\bar{p}) \in \beta'$.*

In other words, a refined success typing is a success typing under some additional constraints. More specifically, a refined success typing is a success typing where the domain is restricted to some subtype of the success typing's domain. Since the set of possible inputs to the function gets restricted, the set of its possible outputs may also get restricted.

We recapitulate: the success typing of a function captures the *set of all its possible uses*. By reducing its domain as much as possible by taking information from all call sites into account, we can infer a *restricted set of uses* which reflects how the function is actually used in a program.

6.2 Module system to the rescue

In Erlang, unlike in e.g. Prolog, the module system cannot be bypassed; all functions have to be part of some module. This provides a way of encapsulating and protecting functions from arbitrary uses. An Erlang module is a `letrec`-style declaration with some additional information. Part of this information is the module's interface: a declaration of functions that are *exported*. The exported functions can be called from any other module, but the non-exported ones can only be called from inside the module. However, since Erlang has higher order functions, non-exported functions can be exposed to the outer world as higher order functions in the form of closures. If a closure is returned by an exported function or if it is passed as an argument to a function in another module, we say that the function represented by the closure *escapes* the module. All exported functions trivially escape since they are exposed to the outer world through the module's interface. Functions that do not escape are called *internal* (or *module-local*) functions. Escaping functions are identified using the escape analysis of Carlsson et al. [5].

In Figure 4 the functions from the example in the beginning of this section are declared in a module, called `my_list_utils`. This module has one escaping function, `length_2`, and one internal function, `length_3`. Since `length_2` escapes, we can make no assumptions about what types it is called with other than what is reflected by its success typing. However, notice that since we know all call sites for the internal function `length_3`, we have the opportunity to refine its domain. By applying the algorithm of Section 5 we find that the success typing of `length_3` is $(list(), any()) \to any()$. This tells us that the first argument in all calls must be a subtype of $list()$ but the second argument can be anything. The partial solutions in the constraint solving phase correspond to type environments of clauses in the original program. For example, we can find the partial solution that corresponds to the second clause of `length_3`, and from this we find the types of the parameters. The first type is $list()$ because of the function call itself, and the second type is $number()$ since it is the result of an addition. From the call in `length_2` we once again find $list()$ for the first argument, and since the second argument is a constant, we find the singleton type `0`. Since the refined domain of a function must include all possible calls, we take the union of the types at the call sites. We can now conclude that the domain of `length_3` can be refined to $list() \cup list() = list()$ in the first argument, and $0 \cup number() = number()$ in the second argument. This information would indeed make it possible to refine the success typing, but as we will see below, it can be refined even more.

6.3 Refining success typings using dataflow analysis

As described in Section 5.3, the inference of success typings works in a bottom-up fashion over the function call graph, propagating information from callees to callers. Since we are now interested in the information flow from caller to callee, it is suitable to use an analysis that propagates information forward in the control flow. We will not describe this analysis in detail, but we will try to give an intuition of how it works.

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

length_2(List) -> length_3(List, 0).

length_3([], N) -> N;
length_3([_|T], N) -> length_3(T, N+1).
```

Figure 4. The module `my_list_utils`

Recall that a success typing states that for a call to succeed, the arguments must be in the expressed domain or the call will surely fail. The dataflow analysis uses this fact to exclude calls that will surely fail from the analysis. For example, assume a function `foo` with success typing $(integer()) \to integer()$, and we find a call

```
..., X = foo(Y), ...
```

somewhere in the program. If the type of `Y` is $atom()$ we know for sure that this call will fail and we can stop the analysis of this program trace. If the type of `Y` is $1 \cup 2 \cup atom()$, then we know that after the call to `foo`, the type of `Y` must be $1 \cup 2$ or the call would have failed. This is also the type that gets propagated to the entry point of `foo` since we already know that a call with $atom()$ will surely fail. If `foo` has already been analyzed for this input type, we get the corresponding return type which is then assigned as the type of `X`. Otherwise, we add the function to the worklist and suspend the analysis of the current trace until we have analyzed `foo`.

The dataflow analysis starts at the entry point of all escaping functions. We can jump-start the analysis by assigning to the arguments of the escaping functions the domain types of the success typings. This is safe since we know that these include all possible inputs for which the functions can return. The information is then propagated forwards in the control flow. At local function calls, the parameter types are propagated in the manner described in the example above. The dataflow analysis is *path independent* inside function bodies, e.g., the type of `case` expressions is collapsed to the union of the clauses. For function calls we have a limited path dependency. We analyze functions for the exact call types up to a limited number of distinct call types. When the number of call types reaches a limit, we widen the call type to the union of all call types. For example, assume that the function `foo` above has three call sites and the argument types are $1 \cup 2$, $3 \cup 4$, and $integer()$ respectively. If we allow for three distinct call types, `foo` will be analyzed for each of the input types, yielding possibly different output types. If we allow only for two call types, the input type of `foo` will be widened to $integer()$ and the return type will be taken from this input type.

Let us revisit the module `my_list_utils` in Figure 4. The dataflow analysis starts with function `length_2`, which is the only escaping function. Since its success typing is $(list()) \to any()$, the type of the variable `List` is $list()$. At the function call we propagate the argument types $list()$ and `0` to `length_3` and since this call pattern has not yet been analyzed, we add it to the worklist and suspend this analysis path. The analysis moves on to `length_3` and assigns the propagated types to arguments at the entry point. Let us first focus on the second clause. The variable `T` is assigned the type $list()$ since taking the tail of a list produces another list. The variable `N` gets assigned the type `0`. At the recursive call the argument types $list()$ and `1` are discovered. Since the function has not yet been analyzed for these input types, this call is added to the worklist and this analysis path is temporarily suspended. The analysis then iterates over `length_3` in this manner until the input type of the second argument reaches the union limit and gets widened to $integer()$. There the widening ends since from now on the function can only be called with subtypes of $integer()$ in the second argument. By analyzing the function using this input type, the return type is found to be $integer()$. Finally, the return types are propagated and the refined success typings are $(list(), integer()) \to integer()$ for `length_3` and $(list()) \to integer()$ for `length_2`. Note that even though the function `length_2` is escaping and we could not refine the domain of its success typing, its range was refined since the refined success typing of `length_3` has a more refined range.

In effect, what the refinement of the success typings does is a *type specialization* of all non-escaping functions based on information which manifests their intended uses: the types of all their calls. Rather than performing function cloning though, at the end of the refinement process, the refined success types for all call patterns are unioned. This can clearly be seen in the following example. The success typing of the `f` function gets refined to reflect its intended uses, which all are of type $float()$ in this module. For `n` a similar process does not occur since this function is called both with integers and floats. Notice however that the range of the `t` function has been refined and accurately reflects the type of its result.

```
-module(arith).
-export([t/1]).

t(N) ->
  X = f(3.14) + N,
  n(42) + n(X).

n(N) -> N + 1.

f(N) -> N + 2.
```

Success typings
`t` :: $(number()) \to number()$
`n` :: $(number()) \to number()$
`f` :: $(number()) \to number()$

Refined success typings
`t` :: $(number()) \to float()$
`n` :: $(number()) \to number()$
`f` :: $(float()) \to float()$

6.4 Current experiences

A significantly weaker and much more ad hoc static analysis than the one we describe in this paper has been used in the publicly available Dialyzer [10] defect detection tool for a period of more than two years now. That analysis, based on a forward dataflow analysis similar to the one of the previous section, has identified literally hundreds of bugs in well-tested, commercial applications of sizes ranging from several thousands to more than a million lines of code.

The current analysis, planned to be integrated into Dialyzer, has so far been used to analyze the Erlang/OTP system and its standard libraries. Besides finding bugs, we intend to automatically annotate all functions of standard libraries with their refined success typings. A separate tool for this task, called TypEr, currently exists in its beta version and can be obtained from the authors. The analysis used in TypEr is the one described in this paper. The analysis is scalable and reasonably fast. On the four-year old laptop of one of the authors, the complete set of the Erlang/OTP standard libraries (amounting to about 700,000 lines of code) is analyzed in half an hour. For comparison, on the same laptop, the BEAM bytecode compiler needs roughly twelve minutes to compile all these files to bytecode and the HiPE native code compiler needs roughly half an hour to compile this bytecode to native code.

7. Concluding Remarks

Changing the philosophy of a programming language, especially one with existing applications of considerable size, is not a task with a high likelihood of success. In this paper, rather than starting from a static type system and trying to squeeze Erlang into it, we followed a different approach. We introduced the concept of success typings into the functional world. Success typings provide an optimistic and totally liberal way of looking at type inference and allow us to uncover the implicit type information which exists in programs, automatically document function interfaces, and detect *definite* type clashes in a flexible and scalable way. Flexible, because success typings allow derivation of type information without type declarations and even in the absence of certain program components. Scalable, because the inference of success typings follows a compositional, bottom-up algorithm which is modular and appears to scale well in practice.

As mentioned, success typings will never miss a function's intended use or report a type violation which only reflects a weakness of the type system. On the other hand, no matter how true, this statement is relatively weak because it is trivially satisfied by success typings which contain no type information. We have shown that by employing constraint-based type inference and by taking advantage of the module system of the language, the success typings get naturally refined, often become quite precise and accurately describe a function's intended use. All this is done *without* sacrificing readability of the typings which are inferred. The practical benefits of doing so should not be underestimated.

Although success typings form a solid basis for capturing the implicit type information in programs written in any dynamically typed functional language, many directions for improvement still exist. Chief among them is the ability to declare and automatically infer recursive types other than lists, the incorporation of (bounded) quantification into the framework, and the ability to maintain dependencies between the types in the domain and the range of a function. We intend to explore some of these issues.

Acknowledgments

During 2005, this research has been supported in part by VINNOVA through the ASTEC (Advanced Software Technology) competence center as part of a project in cooperation with Ericsson.

References

[1] A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *Functional Programming Languages and Computer Architecture*, pages 31–41, 1993.

[2] A. Aiken, E. L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 163–173. ACM Press, 1994.

[3] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall Europe, Herfordshire, Great Britain, second edition, 1996.

[4] R. Carlsson. An introduction to Core Erlang. In *Proceedings of the PLI'01 Erlang Workshop*, Sept. 2001.

[5] R. Carlsson, K. Sagonas, and J. Wilhelmsson. Message analysis for concurrent languages. In R. Cousot, editor, *Static Analysis: Proceedings of the 10th International Symposium*, volume 2694 of *LNCS*, pages 73–90, Berlin, Germany, June 2003. Springer.

[6] R. Cartwright and M. Fagan. Soft typing. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 278–292. ACM Press, 1991.

[7] P. Deransart, M. V. Hermenegildo, and J. Maluszyński, editors. *Analysis and Visualization Tools for Constraint Programming, Constraint Debugging (DiSCiPl project)*, volume 1870 of *LNCS*. Springer, 2000.

[8] R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, Mar. 2002.

[9] M. V. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Integrated program debugging, verification, and optimization using abstract interpretation (and the Ciao system preprocessor). *Sci. Comput. Programming*, 58(1-2):115–140, 2005.

[10] T. Lindahl and K. Sagonas. Detecting software defects in telecom applications through lightweight static analysis: A war story. In C. Wei-Ngan, editor, *Programming Languages and Systems: Proceedings of the Second Asian Symposium (APLAS'04)*, volume 3302 of *LNCS*, pages 91–106. Springer, Nov. 2004.

[11] S. Marlow and P. Wadler. A practical subtyping system for Erlang. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 136–149. ACM Press, June 1997.

[12] S.-O. Nyström. A soft-typing system for Erlang. In *Proceedings of ACM SIGPLAN Erlang Workshop*, pages 56–71. ACM Press, Aug. 2003.

[13] A. Wright and R. Cartwright. A practical soft type system for Scheme. *ACM Trans. Prog. Lang. Syst.*, 19(1):87–152, Jan. 1997.

A. Some Additional Experiences

We provide some additional information to show how inferring success typings can help a programmer to understand better the properties of code that she writes.

A.1 Splitting a list

Consider the `split` function of the Erlang/OTP standard library `lists`. Its implementation is shown in Figure 5. It is slightly obfuscated by the fact that it has to throw a `'badarg'` rather than a `'badmatch'` exception when its arguments do not have the appropriate types. Its online documentation till recently read:

```
split(N, List1) -> {List2, List3}
  Types:
    N = 1..length(List1)
    List1 = List2 = List3 = [term()]
```

Splits `List1` into `List2` and `List3`. `List2` contains the first `N` elements and `List3` the rest of the elements.

There is an obvious discrepancy between the function's code and its documentation: `N` should start from `0` rather than `1`.[5] The slightly more subtle discrepancy is that this function also accepts improper (i.e., not `[]`-terminated) lists as its second argument. The refined success typing for `split/2` that our analysis discovers reads:

$$(integer(), possibly\_improper\_list()) \rightarrow \{[any()], any()\}$$

This typing is correct. Indeed, the call `lists:split(2,[a,b|c])` returns `{[a,b],c}` in Erlang. For some, it may not be easy to comprehend this success typing and see how it is related to `split`'s documentation. The documentation is confusing because of its "`length(List1)`" part: in Erlang the `length` function only works for proper lists. Staring at the code of the `split` function does not help too much either, especially if one is not aware — or has succeeded in forgetting — that, for efficiency, the `is_list` guard of Erlang does not check that its argument is a list, but instead checks whether its top-level constructor is a cons cell or `[]`.

Besides showing the intricacies of inferring success typings in Erlang, this example shows that

1. it is very dangerous to automatically generate type signatures from comments or documentation
2. it is very difficult to impose a static type system that makes assumptions such that e.g. all lists are proper, which are invalid in current Erlang practice.

A.2 Compiling stuff

The HiPE compiler can either compile the bytecode of a whole module or a single function to native code. In Erlang functions are commonly known as MFAs. These are triples which consist of a module name, function name and arity fields. For example, the function of Figure 5(a) is denoted as `{lists,split,2}`.

[5] This typo was fixed in March 2006.

```
split(N, List) when is_integer(N), N >= 0, is_list(List) ->
    case split(N, List, []) of
        Fault when is_atom(Fault) ->
            erlang:error(Fault, [N,List]);
        Result ->
            Result
    end;
split(N, List) -> erlang:error(badarg, [N,List]).
```

(a) The `split/2` function

```
split(0, L, R) ->
    {lists:reverse(R, []), L};
split(N, [H|T], R) ->
    split(N-1, T, [H|R]);
split(_, [], _) ->
    badarg.
```

(b) A non-escaping function that `split/2` uses

Figure 5. A code fragment of the Erlang/OTP `lists` standard module.

One function of the HiPE compiler, which tries to locate the file containing the bytecode to be compiled, reads as shown below.

```
beam_file({M,F,A}) ->
  beam_file(M);
beam_file(Module) when is_atom(Module) ->
  case code:which(Module) of
    non_existing ->
      exit({no_file,Module});
    File ->
      File
  end.
```

Obviously, the programmer tried to benefit from some code reuse here. The success typing we infer for this function is the following:

$$\begin{array}{l} atom() \cup \\ \{atom(), \_, \_\} \cup \\ \{\{atom(), \_, \_\}, \_, \_\} \cup \\ \{\{\{\_, \_, \_\}, \_, \_\}, \_, \_\} \;\rightarrow\; atom() \end{array}$$

where with '_' we denote the $any()$ type. Clearly, this is not what the programmer intended. Note that the code is actually correct, but allows for more general uses than it was envisioned for.

The point of this example is that success typings uncover interesting properties of programs and present this information to the programmer in a relatively comprehensive way without rejecting programs unnecessarily. In this particular case, if the programmer wants to statically detect unintended uses of this function or prohibit them during runtime, she can rewrite the code or add some appropriate type guards in the first clause and make its head look either as follows:

```
beam_file({M,F,A}) when is_atom(M) ->
```

or as follows:

```
beam_file({M,F,A}) when is_atom(M),
                        is_atom(F),
                        is_integer(A) ->
```

in which case the intended success typing

$$(atom() \cup \{atom(), atom(), integer()\}) \rightarrow atom()$$

will be inferred by the success typing algorithm.