A subtype system for functional programming

SVEN-OLOF NYSTRÖM, Uppsala University

Subtyping offers a theoretical foundation for the integration of static and dynamic typing. Subtyping system can reason about types that are more general, for example the universal type that can represent any value. However, subtyping systems are usually limited and lack features that one would like to see in a type system for a programming language. They are also tend to be complex and difficult to extend.

This paper presents a new theoretical approach for subtyping based on logic inference. The new approach is simpler and (we hope) easier to extend.

Using this approach, we have developed a subtyping system for Erlang. The implementation checks ordinary Erlang programs (though naturally not all Erlang programs can be typed, and sometimes it is necessary to add specifications of functions and data types). We describe the implementation of the type checker and give performance measurements.

1 INTRODUCTION

Subtyping systems offer a theoretical foundation for the integration of static and dynamic typing. Subtyping system can reason about more general types, for example the universal type that can represent any value. However, subtyping systems are usually limited and lack features that one would like to see in a type system for a programming languages. They are also tend to be complex and difficult to extend.

This paper presents a static type system for Erlang, a functional programming language with dynamic typing. The type system is designed with Hindley-Milner type inference as a starting point, but relies on subtyping to provide a greater flexibility. The type system is *safe*; programs that type should be free from type errors at run-time.

In functional programming, the standard approach to static typing is to use Hindley-Milner type inference (Milner 1978). Hindley-Milner type inference traces its roots to various forms of typed lambda calculus and has many strong points–it is quite simple, is easy to implement, allows parametric polymorphism and is fast in practice. It is used by many functional programming languages (for example SML and Haskell). However, Hindley-Milner type inference has some important limitations. One is that a recursive data type must be defined using constructors that have are specific to that data type. Thus, a constructor cannot be used for different data types.

Our framework is intended to be flexible and extensible. To type Erlang, the framework has been extended with features that allow it to reason about data types where the sets of constructors is overlapping, and there is a mechanism for conversion of constructors.

Like other subtyping systems, the type system generates a set of constraints when typing a program. The constraints capture the problem of typing the

2016. XXXX-XXXX/2016/1-ART1 $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

program–thus the program is typable if and only if there is a solution to the constraint system. Unlike other systems, in the system described here, whether a set of constraints has a solution depends only on the derivation rules of the constraints–no assumptions are made about the domain of types.

We describe an implementation of the type checker and give performance measurements. The implementation checks ordinary Erlang programs (though naturally not all Erlang programs can be typed, and sometimes it is necessary to add specifications of functions and type declarations).

We have two reasons for choosing Erlang: the language is dynamically typed, thus the run-time system is already adopted to a richer range of values. Second, Erlang does not allow side effects that modify data structures. This simplifies the design of the subtyping system. Our algorithm for type inference does not extract type information for function definitions in a human-readable format, instead the checker compares function definitions to specifications and conversely, that functions are used according to specification.

The rest of this paper is organised as follows. In Section 2 we give an overview of our approach by defining a subtyping system for a simple formal language based on lambda calculus. We describe a simple type checker for this language and discuss how the type system can be extended to manage more powerful constraint languages. Section 3 describes our approach to polymorphism. In Section 4 we extend the subtyping system with constructors, filters (that allow a form of discriminated unions) and conversion of constructors. In Section 5 we discuss the problem of adapting a static type system to Erlang and describe how type declarations and function specifications can be added to Erlang code. Section 6 describes the implementation. Section 7 presents our experimental evaluation and Section 8 places our results in the context of earlier work.

2 HOW TO BUILD A TYPE SYSTEM

Let's start with an overview of the approach. We first develop a simple type checker for a simple functional language and then consider how it can be extended to express more powerful concepts.

2.1 What is a type?

A type expression could be defined as, for example

$$t ::= (t_1 \to t_2) \mid a \mid X$$

where $\to$ builds function types, a represents primitive types, and X ranges over type variables. The idea is that each type expression should evaluate to a type. Now, what is a type? It is possible to use the building blocks of type expressions when building the domain of types, thus the set of types could be defined as follows:

$$T ::= (T_1 \to T_2) \mid a \tag{1}$$

Using a fixed domain of types derived from the syntax of type expressions often works reasonably well and of course saves us the trouble of defining what is meant by a type.

However, sometimes a type domain given by the set of type expressions is not what we want. The most obvious limitation is that a domain of this kind lacks solutions to circular equations such as $X = (t \rightarrow X)$. Now, there have been many attempts to overcome these difficulties by extending the set of types types to include recursive types, see for example (Amadio and Cardelli 1993) but we then lose the simplicity of inductively defined types.

A definition by Barendregt (Barendregt et al. 2013) suggests a different approach:

Definition (Barendregt 11A.1). *A* type structure *is of the form* $S = \langle |S|, \leqslant, \rightarrow \rangle$, *where* $\langle |S|, \leqslant \rangle$ *is a poset and* $\rightarrow : |S|^2 \rightarrow |S|$ *is a binary operation such that for* $a, b, a', b' \in |S|$ *one has*

$$a' \leqslant a \;\&\; b \leqslant b' \Rightarrow (a \rightarrow b) \leqslant (a' \rightarrow b').$$

The structure intended is an *algebraic structure*, i.e., a set of elements with some operations on the set, where the operations should satisfy some algebraic properties. (A poset is a partial order, i.e., the relation $\leqslant$ is transitive, anti-symmetric and reflexive.) The interesting thing is that elements of a type structure do not need to look like type expressions. The properties of Barendregt's type structures are the ones typically seen in subtyping systems; the $\leqslant$ relation defines a partial order, and the arrow operation (which takes two types and builds a new type) satisfies the rule

$$\frac{a' \leqslant a \quad b \leqslant b'}{(a \rightarrow b) \leqslant (a' \rightarrow b')}$$

In other words, this definition says that any combination of a set $|S|$ with an ordering $\leqslant$ and a binary operation $\rightarrow$ over $|S|$ that satisfies the properties is a type structure.

The inductive definition of types earlier (1) (together with some appropriate definition of $\leqslant$) satisfies the definition of type structures. However, there are other interesting type structures, for example type structures with infinite types.

Now, it would be useful to show that it is possible to type the program using *some* type structure, even if we did not know precisely *which* type structure. This is the approach taken in this paper. But first we need to refine the definition of type structures.

For our purposes, the axioms in Barendregt's definition are insufficient as there would not be any program that did *not* type. This would in turn make the problem of type checking rather uninteresting. To remedy this we introduce a set of *atomic types*, types that are distinct from each other and from function types.

Let T, U range over types. Assume a set Atom of atoms, and let A, B range over types associated with these values. Also require that if two function types $T \rightarrow U$ and $T' \rightarrow U'$ are related, i.e., if $(T \rightarrow U) \leqslant (T' \rightarrow U')$, then it holds that $T' \leqslant T$ and $U \leqslant U'$. Barendregt calls type structures that satisfy this property *invertible* but we will assume that all type structures satisfy this property.

Definition 2.1. A *type structure* S is an algebraic structure of the form

$$S = \langle |S|, \leqslant, \rightarrow, \mathsf{Atom}, \mathrm{a} \rangle$$

such that

(1) the relation $\leqslant\ \subseteq |S| \times |S|$ is transitive and reflexive,
(2) $(\rightarrow) : |S| \times |S| \rightarrow |S|$ is a binary operation where for types $T, T', U, U' \in S$ we have $(T \rightarrow U) \leqslant (T' \rightarrow U')$ iff $T' \leqslant T$ and $U \leqslant U'$,
(3) Atom is some set,
(4) $a : \mathsf{Atom} \rightarrow |\mathrm{S}|$,
(5) for $A, B \in \mathsf{Atom}$, $A \neq B$, it never holds that $a(A) \leqslant a(B)$, and
(6) for $A \in \mathsf{Atom}$ and types T, U it never holds that $a(A) \leqslant (T \rightarrow U)$ or $(T \rightarrow U) \leqslant a(A)$.

2.2 Simple constraints

Let $X, Y \in \mathsf{TVar}$ be the set of *type variables.* Also let $A, B \in \mathsf{Atom}$ be the set of *atomic types.* Let the set of *type expressions* $t, u, v, w \in \mathsf{TExp}$ be defined as follows:

(1) $\mathsf{TVar} \subseteq \mathsf{TExp}$,
(2) $A \in \mathsf{TExp}$, if $A \in \mathsf{Atom}$.
(3) $(t_1 \rightarrow t_2) \in \mathsf{TExp}$, if $t_1, t_2 \in \mathsf{TExp}$,

Let the set of *constraints* $\varphi \in \mathsf{Constraint}$ be formulas of the following forms (where $\perp$ is the inconsistent constraint):

(1) $t_1 \leqslant t_2$, for $t_1, t_2 \in \mathsf{TExp}$
(2) $\perp$

A *constraint system* G is a set of constraints.

We express the properties of type structures as derivation rules for constraints; that the $\leqslant$ relation is reflexive and transitive, properties of the $\rightarrow$ operator, and things that must *not* occur, for example $A \leqslant (t \rightarrow u)$ for some atomic type A and arbitrary types t and u.

To describe situations which must not occur, we use $\perp$ to indicate inconsistency, for example in rule AW. Now, it should be easy to see that the derivation rules for constraints of Figure 1 correspond exactly to the axioms given for type structures in Definition 2.1.

We say that a constraint system G is *consistent* if $G \vdash \perp$ does not hold. Naturally we are only interested in consistent constraint systems. We would not expect to find a solution for a constraint system containing, say, a constraint $A \leqslant (t \rightarrow u)$.

2.3 Some mathematical logic

To solve constraint systems, or, more precisely, to determine whether a constraint system can be solved, we will turn to mathematical logic. Mathematical logic is a complex subject and we will only mention some basic definitions and results. We will be brief as details are not important for the rest of the paper. Textbooks on the subject will provide further information, see for example van Dalen (2013).

$$\frac{\phi \in G}{G \vdash \phi}(\in)$$

$$\frac{}{G \vdash t \leqslant t}\ (\mathrm{R}) \qquad \frac{G \vdash t \leqslant u, \qquad G \vdash u \leqslant v}{G \vdash t \leqslant v}\ (\mathrm{T})$$

$$\frac{G \vdash t' \leqslant t, \qquad G \vdash u \leqslant u'}{G \vdash (t \to u) \leqslant (t' \to u')}\ (\mathrm{W})$$

$$\frac{G \vdash (t \to u) \leqslant (t' \to u')}{G \vdash t' \leqslant t}\ (\mathrm{WL}) \qquad \frac{G \vdash (t \to u) \leqslant (t' \to u')}{G \vdash u \leqslant u'}\ (\mathrm{WR})$$

$$\frac{G \vdash A \leqslant (t \to u)}{G \vdash \bot}\ (\mathrm{AW}) \qquad \frac{G \vdash (t \to u) \leqslant A}{G \vdash \bot}\ (\mathrm{WA})$$

$$\frac{G \vdash A \leqslant B \qquad A \neq B}{G \vdash \bot}\ (\mathrm{AA})$$

Fig. 1. Derivation rules for constraints. The rules define the relation $\vdash$.

In predicate logic, a *sentence* may be composed of predicate symbols and expressions (as the constraints defined earlier). A sentence may also be composed using the usual connectives ($\wedge$, $\vee$ and $\neg$) and existential and universal quantifiers. As for constraint systems, we say that a set of sentences is *consistent* if a contradiction cannot be derived.

The rules of Figure 1 can be expressed as sentences, for example rule (T) can be stated

$$\forall XYZ.X \leqslant Y \wedge Y \leqslant Z \implies X \leqslant Z.$$

A constraint system can of course also be viewed as a set of sentences. Thus, the combination of the derivation rules and a constraint system G forms a set of sentences. It should be clear that if a constraint system is consistent, then the corresponding set of sentences is also consistent.

A *structure* is a set of values with a set of symbols; constants (that map to values), functions (that map to operations on the set) and relation symbols over the set (van Dalen 2013, Section 3.2). It should be easy to see that a type structure is also a structure. A structure is said to *model* a set of sentences if each of the sentences holds in the structure (van Dalen 2013, Section 3.4).

In the context of constraint solving, a solution to a constraint system corresponds to a structure that models a set of sentences. If we want to determine whether a constraint system can be solved, but we are not interested in the details of the solution, we can use a result by Henkin, known as the *model existence property* (Henkin 1949), see also (van Dalen 2013, Section 4.1). Henkin shows that for a consistent set of sentences, it is possible to construct a structure that models the set of sentences.

In other words, if a constraint system is consistent, there is some type structure for which it has a solution. In Section 2.7 we show that there is a straight-forward algorithm for checking the consistency of a constraint system. If the algorithm finds that the constraint system is consistent, the program types.

2.4 Lambda calculus

We first develop a system for subtype inference for lambda calculus. We will later look at variations of lambda calculus extended with important features of Erlang and discuss how they can be typed.

Why lambda calculus? Lambda calculus is a simple and efficient formalism. Using a simple formalism will simplify reasoning and reduce clutter. Lambda calculus is close to functional programming, and particularly suited for reasoning about types.

This presentation is intended to provide an overview for readers who are not familiar with lambda calculus, focusing on features of lambda calculus that differ from theoretical models of functional programming languages.

We extend lambda terms to include terms that represent atoms. Given a set of variables $x \in \mathsf{Var}$ and a set of atoms $A \in \mathsf{Atom}$, the set of lambda terms is inductively defined as:

$$M ::= x \mid M_1 M_2 \mid \lambda x.M_1 \mid A$$

We will let the variables M, N, P range over lambda terms. A term which is an atom will have the atom as type.

We say that an occurrence of a variable in a lambda term is *free* if it is not "bound" by a lambda. For example, the variable x is free in the terms $\lambda y.x$ and xz but bound in $\lambda x.x$. There are terms in which one occurrence of a variable is free and another is bound, for example $x(\lambda x.x)$.

Now, one important feature of lambda calculus which is easy to overlook is that terms are considered to be equivalent up to renaming of bound variables, for example, the terms $\lambda x.x$ and $\lambda y.y$ represent the same function (the identity function). Thus a lambda term may have many syntactic representations. In the typing of a term M, we will assume that the representation of M is chosen such that any free variable is not also bound, and no variable is bound in more than one sub-term.

We will need a substitution operator. For a variable x and terms M and N, let $M[x := N]$ be the result of replacing the variable x with the term N in M. The reader may have noted a potential pitfall; if a free variable of N becomes bound in the substitution the result is probably not the intended one. This can be avoided with a careful definition of substitution, but to keep things simple we will simply assume that in a substitution, the representation of M is chosen so that no free variable of N becomes bound.

The semantics of lambda calculus can now be expressed using a single reduction rule:

$$(\lambda x.M)N \longrightarrow_\beta M[x := N] .$$

A lambda term is said to be a *redex* if it can be on the left hand side in this rule above, i.e., if it is of the form $(\lambda x.M)N$. Clearly, for any redex M there is a lambda term M' such that $M \longrightarrow M'$.

We say that $M \longrightarrow M'$ if M' if there is some sub-term N of M such that $N \longrightarrow_\beta N'$, and M' is the result of replacing one occurrence of N in M with

$$\frac{(x : t) \in \Gamma}{\Gamma \Vdash x : t} \qquad \text{(axiom)}$$

$$\frac{\Gamma \Vdash M : t \to u \qquad \Gamma \Vdash N : t}{\Gamma \Vdash MN : u} \qquad \text{(application)}$$

$$\frac{\Gamma \cup \{x : t\} \Vdash M : u}{\Gamma \Vdash \lambda x.M : t \to u} \qquad \text{(abstraction)}$$

$$\frac{}{\Gamma \Vdash A : A} \qquad \text{(atom)}$$

$$\frac{\Gamma \Vdash M : t \qquad t \leqslant u}{\Gamma \Vdash M : u} \qquad \text{(subsumption)}$$

Fig. 2. Subtyping rules.

N'. We write $M \longrightarrow\!\!\!\!\!\rightarrow N$ if there is a sequence

$$M_1 \longrightarrow M_2 \longrightarrow \ldots \longrightarrow M_n$$

with $M = M_1$ and $N = M_n$.

2.5 Typing lambda calculus

An *environment* Γ is a set $\{x_1 : t_1, \ldots, x_n : t_n\}$ where the x_i are distinct variables, and the t_i are type expressions. For a variable x, an environment should contain at most one binding $x : t$ of x.

A typing is written $\Gamma \Vdash M : t$ and indicates that the lambda term M has the type t in environment Γ. (We use the symbol $\Vdash$ for typings to reduce the risk of confusion between derivations in the constraint system and typings.) If Γ is empty, we will sometimes write $\Vdash M : t$.

The type rules are given in Figure 2. The first three type rules are the standard rules of simply typed lambda calculus (see for example (Barendregt et al. 2013, Figure 1.6)). The subsumption rule says simply that any type can be replaced with a more general type (Mitchell 1991)

It is often convenient to make the constraints of a typing explicit, thus we will sometimes write $\Gamma \Vdash M : t\ [G]$ to indicate that the typing $\Gamma \Vdash M : t$ holds, provided that the constraint system G can be solved. Naturally, whenever $\Gamma \Vdash M : t$ there is some constraint system G such that $\Gamma \Vdash M : t\ [G]$. For the reader's convenience we show the rules on this format in Figure 3.

From a practical point of view the subsumption rule poses some difficulties as it can be inserted anywhere in the derivation of a typing. The other rules are associated with different ways of building terms, so that the tree shape of the derivation of a typing for a term is given by the term. Now, since the subtyping relation is reflexive and transitive there is for any derivation of a typing an equivalent derivation where every other rule is an application of the subsumption rule. In other words, it is always possible to find a derivation of the typing with a shape that is given by the term. This is discussed in more detail by Kozen et al. (1994) and Palsberg and O'Keefe (1995).

$$\frac{(x : t) \in \Gamma}{\Gamma \Vdash x : t \; [\emptyset]} \qquad \text{(axiom)}$$

$$\frac{\Gamma \Vdash M : t \to u \; [G_1] \qquad \Gamma \Vdash N : t \; [G_2]}{\Gamma \Vdash MN : u \; [G_1 \cup G_2]} \qquad \text{(application)}$$

$$\frac{\Gamma \cup \{x : t\} \Vdash M : u \; [G]}{\Gamma \Vdash \lambda x.M : t \to u \; [G]} \qquad \text{(abstraction)}$$

$$\frac{}{\Gamma \Vdash A : A \; [\emptyset]} \qquad \text{(atom)}$$

$$\frac{\Gamma \Vdash M : t \; [G]}{\Gamma \Vdash M : u \; [G \cup \{t \leqslant u\}]} \qquad \text{(subsumption)}$$

Fig. 3. Subtyping rules with explicit constraint systems.

$$\begin{aligned}
&w_\Gamma(x) = \langle \emptyset, \Gamma(x) \rangle \\
&w_\Gamma(M_1 M_2) = \langle G_1 \cup G_2 \cup \{t_1 \leqslant (t_2 \to X)\}, X \rangle \\
&\qquad \text{where } X \text{ is a fresh type variable,} \\
&\qquad\qquad \langle G_1, t_1 \rangle = w_\Gamma(M_1), \text{and} \\
&\qquad\qquad \langle G_2, t_2 \rangle = w_\Gamma(M_2) \\
&w_\Gamma(\lambda x.M_1) = \langle G_1, X \to t_1 \rangle \\
&\qquad \text{where } Y \text{ is a fresh type variable,} \\
&\qquad\qquad \Gamma_1 = \Gamma \cup [x : Y], \text{and} \\
&\qquad\qquad \langle G_1, t_1 \rangle = w_{\Gamma_1}(M_1) \\
&w_\Gamma(A) = \langle 0, A \rangle
\end{aligned}$$

Fig. 4. Explicit construction of constraint system

We can use this insight in an explicit construction of the constraint system that needs to be solved in order to type the term. For a type environment Γ and a term M, compute a pair of a constraint system and a type expression: $w_\Gamma(M) = \langle G, t \rangle$. The constraint system G has a solution exactly in the cases M can be typed (Figure 4). Thus the constraint system necessary to type a lambda-term can be constructed by a straight-forward traversal of the term. The term types exactly when the constraint system has a solution.

2.6 Safety

A desirable property of a type system is *safety.* This is usually taken to mean that if a program types, certain errors should not occur at run time. Milner (1978) shows that a program that types is "semantically free of type violation", i.e., that "for example, an integer is never added to a truth value or applied to an argument". One way to show this property is via the *subject reduction property.*

The subject reduction property states an invariant for typings; if M is a term that types, that is, $\Gamma \Vdash M : t$, for some environment Γ and type expression t, and M reduces in one or more steps to some other term ($M \longrightarrow\!\!\!\!\!\rightarrow N$) then that term will have the same type, $\Gamma \Vdash N : t$. If N is a term that cannot type, for example an application of an arithmetic operation to strings, then the subject reduction property guarantees that no term that types can be reduced to N. (The word "subject" refers to the term M in a typing $\Gamma \Vdash M : t$.)

Lemma 2.2. *If $M \longrightarrow\!\!\!\!\!\rightarrow N$ and $\Gamma \Vdash M : t\ [G]$ then $\Gamma \Vdash N : t\ [G]$.*

The original proof of the subject reduction property for lambda calculus was given by Curry and later extended to subtyping by Mitchell (1984, 1991). See also Barendregt et al. (2013, Section 1.2 and 11.1). We will not repeat the proofs here.

Wright and Felleisen (1994) suggested that in addition to the subject reduction property it would also be useful to show a property sometimes called "progress"; that any term that types in an empty environment is either a value or can be reduced to another term. See also Pierce (2002, Section 8.3). In our extended lambda calculus the values would be atoms and abstractions (terms of the form $\lambda x.M$). To show progress a simple inductive argument on M will suffice.

2.7 Checking that a program types

A lambda term M types if there is some derivation of the typing $\Vdash M : t\ [G]$, where the constraint system G has a solution. By the model existence property (Section 2.3) it is sufficient to show that the constraint system G is consistent. We will now describe an algorithm for checking consistency of a constraint system.

Definition 2.3. Given a constraint system G, define $(G)_n$, for $n \geq 0$, to be the smallest sets that satisfy the following:

(1) $(G)_0 = G$,
(2) for all n, $(G)_{n+1} \supseteq (G)_n$,
(3) for all even $n > 0$, if the constraint $(t \to u) \leqslant (t' \to u')$ is in $(G)_{n-1}$, then the constraints $t' \leqslant t$ and $u \leqslant u'$ are in $(G)_n$, and
(4) for all odd $n > 0$, if the constraints $t \leqslant X$ and $X \leqslant u$ are in $(G)_{n-1}$, then $(t \leqslant u) \in (G)_n$.

Let $G^* = \bigcup_n (G)_n$.

The complexity of constructing G^* can be determined by a simple argument (Heintze 1994). First, note that G^* only contains type expressions present in G. Thus if the size of G is n, and G contains no more than n expressions, there are less than n^2 inequalities in G^*, which sets a bound to the space used by the construction. When an inequality $t \leqslant u$ is added to the constraint system, the algorithm must examine inequalities of the forms $t' \leqslant t$ and $u \leqslant u'$ (in the *odd* step). This may, at worst, require work proportional to the number of expressions in G, thus the cost of adding one constraint is $O(n)$ and the worst-case complexity of the algorithm is $O(n^3)$.

(Cubic worst-case complexity is actually a *good* result. Many widely used algorithms in programming language implementation have worse complexity but perform well for typical programs. The results by Heintze and Tardieu

(2001) on efficient implementation of a form of points-to analysis for an imperative language are particularly promising as that form of points-to analysis shares many properties with the computation of G^*.)

The definition of G^* might seem unnecessarily restrictive as it would not add to the complexity of computing G^* if Item 4 of the definition was generalised to allow arbitrary expressions instead of a variable. However, it turns out that this seemingly straight-forward change would make the proof of Theorem 2.4 more complicated, in particular Proposition 2.7 would need to be restated.

We say that a constraint system is *locally consistent* if G^* does not contain any immediately inconsistent constraints such as $\bot$, $A \leqslant (t \to U)$, $(t \to U) \leqslant A$, or $A \leqslant B$, for distinct atoms A and B. It turns out that local consistency coincides with consistency.

Theorem 2.4. *A constraint system G is consistent iff G is locally consistent.*

It should be clear that a consistent constraint system is also locally consistent. To show the converse, that a locally consistent constraint system is consistent, we consider the proof rules of Figure 1. The question we need to ask is: If we can deduce $G \vdash \varphi$ in a single application of one of the rules, how will $(G \cup \{\varphi\})^*$ differ from G^*?

Rules ($\in$), (WL) and (WR) are applied in the computation of G^*, so if $G \vdash \varphi$ can be deduced using one application of one of these rules we have $\varphi \in G^*$. As G is assumed to be locally consistent the rules (AW), (WA) and (AA) can be excluded as their use implies that G is *not* locally consistent.

We next consider the remaining rules (R), (T) and (W) and show that while they add new constraints, local consistency will not change. We state properties of these derivation rules in the following propositions. They can be shown by induction over n.

Proposition 2.5 (Rule R). *Suppose that G is a constraint system, t some type expression and φ an inequality. Let $H = G \cup \{t \leqslant t\}$.*

Whenever $\varphi \in (H)_n$ it holds either that

(1) *$\varphi \in (G)_n$, or*
(2) *$\varphi = (u \leqslant u)$, some subexpression u of t.*

Proposition 2.6 (Rule T). *Suppose that G is locally consistent and contains the constraints $t \leqslant t'$ and $t' \leqslant t''$. Let $H = G \cup \{t \leqslant t''\}$.*

Whenever a constraint $u \leqslant v$ occurs in $(H)_n$, there are type expressions $w_1, w_2, \ldots, w_m$ and an integer k such that $w_1 = u$, $w_m = v$, and the constraint $w_i \leqslant w_{i+1}$ occurs in $(G)_k$, for $i < m$.

Proposition 2.7 (Rule W). *Let G be a constraint system containing the constraints $t \leqslant t'$ and $u \leqslant u'$. Let $\varphi = ((t' \to u) \leqslant (t \to u'))$ and $H = G \cup \{\varphi\}$. It follows that whenever a constraint ψ occurs in $(H)_n$, we have either*

(1) *$\psi \in (G)_n$, or*
(2) *$\psi = \varphi$.*

The proof of the theorem uses these propositions to show that a sequence of applications of the derivation rules R, T and W to a locally consistent constraint system cannot lead to the derivation of $\bot$, thus if a constraint system is locally consistent it is also consistent.

2.8 How to extend the constraint language

In our framework, introducing new forms of type expressions is entirely unproblematic, since without any derivation rules that operate on them, it is not possible to use the new expressions to prove new things. Adding derivation rules is a different matter. A new derivation rule allows us to draw new conclusions, thus it could cause a previously consistent constraint system to become inconsistent. We will consider a simple example; the addition of a universal type. We will show how universal types can be accommodated in our framework.

We use the symbol 1 for the type expression that represents the universal type. The additional rules are stated in Figure 5

Rule (U) states that 1 is the greatest type according to the subtyping order. For any type t, we can conclude that t is a subtype of 1. Rules (UW) and (UA) state that no type given by an atom expression or an arrow expression may be greater than the universal type. More explicitly, if a constraint which states that the universal type is a subtype of (for example) an atomic type is encountered a contradiction can be derived. To handle these rules in our framework, we define constraints of the forms $1 \leqslant (t \to u)$ and $1 \leqslant A$ to be immediately inconsistent. We also need to show that rule (U) preserves local consistency.

Generally speaking our framework can be extended to accommodate new derivation rules if they fall into one of three categories:

(1) Rules that describe situations where inconsistency follows from a constraint. Such constraints can be included in the set of immediately inconsistent constraints, provided that it is possible to implement a constant-time test that recognises them. In our example Rules (UW) and (UA) fall into this category,
(2) Rules that preserve local consistency. Our example has one such rule; Rule (U).
(3) Rules that require extending the computation of G^*. Such rules must not introduce new type expressions (as that could affect complexity and might even cause the computation to loop). We have seen one rule that falls into this category: Rule (W) of Figure 1.

$$\frac{}{t \leqslant 1} \tag{U}$$

$$\frac{1 \leqslant (t \to u)}{\perp} \tag{UW}$$

$$\frac{1 \leqslant A}{\perp} \tag{UA}$$

Fig. 5. Derivation rules for the universal types.

2.9 Workflow in the design of a type system

$$\begin{tikzcd}
\text{Reduction rules} \arrow[dr] & \text{Type rules} \arrow[d] & \text{Constraint rules} \arrow[dl] \arrow[dr] & \\
& \text{Safety} & & \text{Algorithm for checking consistency}
\end{tikzcd}$$

The dependencies are summarised in the diagram above. If the reduction rules are modified or extended, the derivation rules (of constraints) need to be sufficiently powerful to show safety properties (in particular, the subject reduction property), thus it may be necessary to introduce new constraint rules. A change in the constraint rules may in turn require a change in the constraint checking algorithm. On one hand the derivation rules need to be sufficiently powerful to guarantee the subject reduction property, on the other hand they must not be so expressive that they cannot be implemented efficiently.

3 POLYMORPHISM

Hindley-Milner type checking implements a form of polymorphism which is sometimes known as *parametric polymorphism.* A function defined in a let-expression, for example

```
let f x = ...
in
  ...
end
```

is treated as polymorphic. This means that any occurrence of `f` in the body of the let is typed independently of the other occurrences of `f`. (In a sequence of function definitions, any earlier definition is treated as if it was defined in a `let` whose body contains the following function definitions.) Parametric polymorphism is accomplished by representing the type of `f` as a *type scheme* which can be instantiated many times. In Hindley-Milner typing, this type scheme has a minimal form (sometimes referred to as the *principal type*),

For subtyping it seems difficult to find a minimal representation of the information needed to type a let-bound function. We have instead taken a straight-forward approach: extract the constraints needed to type `f`. Then generate a copy of the constraint system for each use of `f` in the body of the function. To reduce the cost of duplicating the constraint system, we apply various simplifications to reduce the size of the constraint system.

Finding minimal representations of constraint systems would help performance. However, Pottier (Pottier 2001) and Flanagan and Felleisen (Flanagan and Felleisen 1999) show that computing an optimal representation of a constraint system for subtyping is related to the problem of minimizing non-deterministic finite automata which is known to be a problem of high complexity. Since the goal of minimisation is to speed up the type checker it seems counter-productive to use an expensive algorithm for minimisation. Both Pottier and Flanagan conclude that a reasonable compromise is the use of faster simplification algorithms which are not guaranteed to produce optimal results.

Surprisingly, Mairson (1990) shows that Hindley-Milner type checking is actually DEXPTIME-hard. In other words, the cost of typing a program is, in the worst case, exponential in the size of the program. Making the very reasonable assumption that the problem of typing a program does not become easier when one goes from equality-based typing to subtyping, this suggests that the straight-forward approach sketched here is actually asymptotically optimal.

4 THE EXTENDED LAMBDA CALCULUS

We extend the simple language of Section 2 to accommodate the Erlang programming language. First, Erlang has a rich set of data type constructors (in contrast to the simple language which only has atoms and functions). A type can be described by a set of constructors, each applied to a number of types. An Erlang program, say something like this:

```
f({leaf, X}) -> ...
f(Y) -> ...
```

can easily distinguish between data built using a particular constructor and data that is *not.* Thus we want to be able to isolate data that does not match a constructor, both in the extended lambda calculus and in the constraint language. In the extended lambda calculus we express this using a special construct, the *open case expression.* The constraint language uses *filters* to separate the part of a type that is built using a particular set of constructors. Filters are also used to reason about discriminated unions. Last, in some cases we need to allow *conversion* between types created using different constructors.

4.1 Constructors

The data types of Erlang include (for example) tuples, lists, atoms, integers and floating point values. In the extended lambda calculus and the constraint language, we model all these constructors uniformly. We assume a set of constructors $c \in C$, where each constructor has an arity. Each argument of a constructor is either covariant or contravariant (the only constructors with contravariant arguments are function types). Constructors form terms in the the extended lambda calculus and type expressions in the constraint language, thus constructors build both data and types.

For the representation of function types we reserve a constructor c_λ with arity 2 that does not occur in any term. The first argument of c_λ is contravariant, the second covariant.

We will always assume that a constructor is used with the correct number of arguments, thus we will often omit reference to the arity of the constructor. We will sometimes refer to a term or a type expression of the form $\langle c \ \ldots \rangle$ as a *constructor term* or a *constructor expression* or more specifically as a c-term or a c-expression.

4.2 Filters and unions

Erlang has a fixed set of constructors that can be used to build recursive data types. This should be contrasted with the situation in programming languages based on Hindley-Milner type checking, where each data type has its own set of constructors.

Consider, for example, the following Haskell data type definition:

```
data Tree = Leaf Integer
          | Branch Tree Tree
```

This type definition introduces the constructors `Leaf` and `Branch` (and they cannot be used to build data structures of any other type). In our system, the corresponding data type might be defined

```
+type tree() = {leaf, integer()}
             + {branch, tree(), tree()}.
```

Here, the data structure uses the tagged tuples `{leaf, ...}` and `{branch, ...}` as constructors. They may of course be used in other parts of the program.

The specification language can express that a type is a union of two types, but there are limitations. Consider an inequality

$$\langle c_1 \ \ldots\rangle \cup \langle c_2 \ \ldots\rangle \leqslant X.$$

A constraint of this form could occur if one wanted to type check an Erlang function that is specified to accept the tree data type as a parameter. Expressing this in the constraint language is easy:

$$\langle c_1 \ \ldots\rangle \leqslant X, \langle c_2 \ \ldots\rangle \leqslant X.$$

However, sometimes we want to put the union on the right-hand side of the inequality. An inequality of this type would take the form:

$$Y \leqslant \langle c_1 \ \ldots\rangle \cup \langle c_2 \ \ldots\rangle. \tag{2}$$

A constraint of this form could occur (for example) when the type checker verifies that a function does indeed return a tree. We will consider the case when the two constructors c_1 and c_2 are distinct.

Instead of adding union types to our constraint language, we introduce a new form of type expressions which we will call *filters.* A constraint that uses a filter takes the form

$$X \upharpoonright S \leqslant t,$$

where S is a set of constructors, X is a type variable and t is a type expression. Filters may only occur on the left-hand side of an inequality. (Applying filters to other type expressions or allowing filters on the right-hand side of $\leqslant$ would not cause any major difficulties but would complicate derivation rules and the formulation of the consistency checking algorithm, so we focus on the case where filters are really useful.)

The idea is that a filter only lets through those subtypes of X that use a constructor which is a member of S. This can be expressed in the following derivation rule:

$$\frac{G \vdash \langle c\ t_1 \ldots t_n\rangle \leqslant X \qquad X \upharpoonright S \leqslant u \qquad c \in S}{G \vdash \langle c\ t_1 \ldots t_n\rangle \leqslant u} \tag{F}$$

Note that this derivation rule fits Category 3 of Section 2.8 as it is straight-forward to extend the algorithm for computing G^* to handle this rule.

Turning back to our example (2), checking that a type belongs to one of the two type expressions $\langle c_1 \ \ldots\rangle$ and $\langle c_2 \ \ldots\rangle$ can be expressed with the constraints

$$Y \upharpoonright \{c_1\} \leqslant \langle c_1 \ \ldots\rangle \quad \text{and} \quad Y \upharpoonright S \leqslant \langle c_2 \ \ldots\rangle$$

where S is the largest set of constructors that does not contain c_1. Thus the first filter will match only those type expressions that use the constructor c_1, but the second filter will match those that do *not* use c_1. (Please recall that we assumed that c_1 and c_2 were distinct.)

4.3 Open case statements

As mentioned, Erlang makes it easy to write code that performs a case analysis on a data structure depending on whether it belongs to one subtype or not. In the extended lambda calculus we express this mechanism through *open case terms.* These take the form

$$\begin{aligned}&\text{case}(M,\\ &\quad \langle c\ x_1 \ldots x_n\rangle \Rightarrow N,\\ &\quad y \Rightarrow P).\end{aligned}$$

The idea is that if the term M matches the pattern $\langle c\ x_1 \ldots x_n\rangle$, the first branch, the term N, is selected. The second branch is only selected when the term does *not* match the pattern. The syntactic form used here was first considered by Heintze (1994) in the context of set-based analysis.

The reduction rules and the type rule for the open case expression will need to take conversion into account, but we first show them without conversion.

The reduction rules for the open case expression are straight-forward, one rule for the case where the matching succeeds, and two for the cases where matching fails because the term is either an abstraction or a term built using a different constructor.

$$\begin{aligned}&\text{case}(\langle c\ M_1 \ldots M_n\rangle, \langle c\ x_1 \ldots x_n\rangle \Rightarrow N, y \Rightarrow P) \longrightarrow\\ &\qquad N\,[x_1 := M_1, \ldots, x_n := M_n]\,,\\ &\text{case}(\lambda x.M, \langle c\ \ldots\rangle \Rightarrow N, y \Rightarrow P) \longrightarrow P\,[y := \lambda x.M]\\ &\text{case}(M, \langle c\ \ldots\rangle \Rightarrow N, y \Rightarrow P) \longrightarrow P\,[y := M]\,,\\ &\qquad \text{where } M = \langle d\ \ldots\rangle \text{and } d \neq c\end{aligned}$$

In the type rule for open case, we use filters to separate the cases where the term matches from the ones where the term does not match the pattern. The type rule becomes rather elaborate:

$$\frac{\begin{array}{cr} \Gamma \Vdash M : t & \\ t \leqslant X & \\ X \upharpoonright \{c\} \leqslant \langle c\, u_1 \ldots u_n \rangle & (3) \\ \Gamma\,[x_1 \mapsto u_1, \ldots, x_n \mapsto u_n] \Vdash N : w & \\ X \upharpoonright C \setminus \{c\} \leqslant Z & (5) \\ \Gamma[y \mapsto Z\,] \Vdash P : w & \end{array}}{\Gamma \Vdash \mathrm{case}(M, \langle c\, x_1 \ldots x_n \rangle \Rightarrow N, y \Rightarrow P) : w} \quad (\mathrm{case})$$

Note the use of filters to extract the subtypes of t that match the constructor (line 3) and those that do not match the pattern (line 5).

Showing the subject reduction property is straight-forward but somewhat tedious. The progress property can be established by extending the induction proof.

4.4 Conversion

Since Erlang was not designed as a typed language from the start, the way constructors are used by applications, libraries and built-in primitives sometimes makes it difficult to determine which attributes of a data structure should be thought of as a constructor. For example, while it is clear that each atom should be its own constructor, there are operations that work on any atom, so one would like a type that represented *all* atoms. We have seen tagged tuples, but sometimes tuples are not tagged (different untagged tuples are only distinguished by their length), thus one would like a type for untagged tuples for each length. Some Erlang primitives treat tuples as arrays, so one would also like a type that describes tuples of any length.

4.4.1 Some Erlang constructors. We will look at the constructors listed in Table 1. (Erlang has other constructors, but the ones listed here are the most interesting.) The constructor $\mathbf{tuple}_a^n$ represents a tuple of length n which is tagged with the atom a. This constructor has arity $n - 1$, as the first element of the tuple is implicit. For untagged tuples of length n we use the constructor $\mathbf{tuple}^n$ which of course has arity n. The constructor **tuple** (of arity 1) is used when a tuple is uniform, i.e., each element of the tuple has the same type. For an atom a, the nullary constructor $\mathbf{atom}_a$ represents that atom, in other words, the term $\langle \mathbf{atom}_a \rangle$ is that atom. The type expression $\langle \mathbf{atom}_a \rangle$ gives us the type consisting of the atom a. The type expression $\langle \mathbf{atom} \rangle$ gives the type of all atoms, and the type expression $\langle \mathbf{any} \rangle$ the universal type.

4.4.2 Specifying conversion. Generally speaking, we need to resolve constraints of the form

$$\langle c\, t_1 \ldots t_n \rangle \leqslant \langle d\, u_1 \ldots u_m \rangle,$$

where the left-hand expression can be converted to the right-hand expression.

In the extended lambda calculus, conversion comes into play in the open case expressions. If the pattern of an open case expression is an untagged tuple, and the term being matched is a tagged tuple, the matching may succeed (if the lengths of the tuples are the same).

Table 1. Some constructors in the Erlang type system. The variable n ranges over non-negative integers and a over Erlang atoms.

Constructor	Description	Arity
$\mathbf{tuple}_a^n$	tagged tuple	$n-1$
$\mathbf{tuple}^n$	untagged tuple	n
$\mathbf{tuple}$	uniform tuple	1
$\mathbf{atom}_a$	a specific atom	0
$\mathbf{atom}$	any atom	0
$\mathbf{any}$	universal type	0

Fig. 6. Conversion over terms

$$\langle \mathbf{tuple}_a^n\ M_2\ \dots\ M_n\rangle \lhd \langle \mathbf{tuple}^n\ \langle \mathbf{atom}_a\rangle\ M_2\ \dots\ M_n\rangle$$
$$\langle \mathbf{tuple}^n\ M\ \dots\ M\rangle \lhd \langle \mathbf{tuple}\ M\rangle$$
$$\langle \mathbf{atom}_a\rangle \lhd \langle \mathbf{atom}\rangle$$
$$t \lhd \langle \mathbf{any}\rangle$$

Fig. 7. Conversion over type expressions

$$\langle \mathbf{tuple}_a^n\ t_2\ \dots\ t_n\rangle \lhd \langle \mathbf{tuple}^n\ \langle \mathbf{atom}_a\rangle\ t_2\ \dots\ t_n\rangle$$
$$\langle \mathbf{tuple}^n\ t\ \dots\ t\rangle \lhd \langle \mathbf{tuple}\ t\rangle$$
$$\langle \mathbf{atom}_a\rangle \lhd \langle \mathbf{atom}\rangle$$
$$t \lhd \langle \mathbf{any}\rangle$$

We start by specifying relations $\lhd$ over terms and type expression. We define these relations as the minimal transitive and reflexive relation which satisfies the properties stated in figures 6 and 7, for arbitrary terms $M, M_1, \dots, M_n$, type expressions $t, t_1, \dots, t_n$ and atoms a.

The definition of $\lhd$ allows conversions such as

$$\langle \mathbf{tuple}^2_{\mathtt{leaf}}\ t\rangle \lhd \langle \mathbf{tuple}^2\ \langle \mathbf{atom}_{\mathtt{leaf}}\rangle t\rangle.$$

In other words, a tagged tuple is also an untagged tuple.

We can now give a subtype rule that allows conversion:

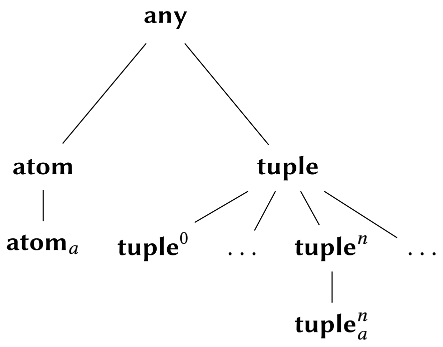
$$\frac{G \vdash t \lhd u}{G \vdash t \leq u}\ (\lhd)$$

According to this rule, an expression t a subtype of u whenever t can be converted to u.

Note however, that Rule ($\lhd$) does not quite fit the type checking algorithm (Section 2.7) as a constraint $t \leqslant u$ may need to be resolved via a combination of the ($\lhd$) rule and a generalization of Rule W of Figure 1. However, it is easy to define a general rule that combines these rules. Given a constraint $t \leqslant u$, where both t and u are constructor expressions, we can find a finite set S of constraints (using only proper sub-expressions of t and u) such that for any constraint system G, $G \vdash t \leqslant u$ iff $G \vdash \varphi$, for all $\varphi \in S$.

We define a function coerce to satisfy:

Fig. 8. Some constructors and how they are related under $\ll$.



(1) coerce $\langle c\ t_1, \ldots, t_n\rangle\langle c\ u_1, \ldots, u_n\rangle = \{\varphi_1, \ldots, \varphi_n\}$, where $\varphi_i = (t_i \leqslant u_i)$, if the ith argument of c is covariant, and $\varphi_i = (u_i \leqslant t_i)$ otherwise,
(2) coerce $\langle \mathbf{tuple}^n_a\ t_2 \ldots t_n\rangle\langle \mathbf{tuple}^n\ u_1 \ldots u_n\rangle = \{\langle \mathbf{atom}_a\rangle \leqslant u_1, t_2 \leqslant u_2, \ldots, t_n \leqslant u_n\}$,
(3) coerce $\langle \mathbf{tuple}^n\ t_1 \ldots t_n\rangle\langle \mathbf{tuple}\ u\rangle = \{t_1 \leqslant u, \ldots, t_n \leqslant u\}$,
(4) coerce $\langle \mathbf{atom}_a\rangle\langle \mathbf{atom}\rangle = \emptyset$.
(5) coerce $t\ \langle \mathbf{any}\rangle = \emptyset$.

For any pairs of constructor expressions t and u, coerce $t\ u$ provides a set of constraints that need to hold in order for the constraint $t \leqslant u$ to hold.

We use the symbol $\ll$ for conversion of constructors, i.e., we write $c \ll c'$ if there are some terms and type expressions $\langle c\ \ldots\rangle$ that can be converted to $\langle c'\ \ldots\rangle$ and write $c \not\ll c'$ when this is not the case. Note that $\ll$ is a partial order, i.e., the relation is transitive, reflexive and anti-symmetric. We say that a constructor c has a *parent* c' if $c \ll c'$ but $c \neq c'$ and whenever $c \ll d$, either $c = d$ or $c' \ll d$. Also note that a constructor has a parent unless it is the universal constructor. Thus the constructors form a tree where the root is the universal constructor (Figure 8),

4.4.3 Conversion of terms. In the extended lambda calculus, conversion comes into play in the open case statements. The conversion relation $M \lhd N$ holds when M and N are constructor terms and M can be converted to N. In a case expression, a term may be converted to fit a pattern.

$$\text{case}(M, \langle c\ x_1 \ldots x_n\rangle \Rightarrow N, y \Rightarrow P) \longrightarrow N\,[x_1 := M_1, \ldots, x_n := M_n]\,,$$
$$\text{where M} \lhd \langle \text{c M}_1 \ldots \text{M}_\text{n}\rangle$$

For example, when the term being matched is a tagged tuple and the pattern is an untagged tuple, we have the conversion

$$\langle \mathbf{tuple}^2_{\mathtt{leaf}}\ M\rangle \lhd \langle \mathbf{tuple}^2\ \langle \mathbf{atom}_{\mathtt{leaf}}\rangle\ M\rangle.$$

4.4.4 Filters and conversion. We will assume that in all constraints $X \upharpoonright S \leqslant Y$ the set S is up-closed, i.e., when $c \ll c'$ and $c \in S$, we also have $c' \in S$.

Consider an example. Suppose S is a non-empty set of constructors not containing the constructor **any** and $t = \langle \mathbf{any}\rangle$. If $G \vdash t \leqslant X$ and $G \vdash X \upharpoonright S \leqslant Y$,

Fig. 9. Derivation rules for extended constraints.

$$\frac{\varphi \in G}{G \vdash \varphi} \quad (\in)$$

$$\frac{}{G \vdash t \leqslant t} \quad (\mathrm{R})$$

$$\frac{G \vdash t \leqslant u, \qquad G \vdash u \leqslant v}{G \vdash t \leqslant v} \quad (\mathrm{T})$$

$$\frac{G \vdash \mathrm{coerce}(t, u)}{G \vdash t \leqslant u} \quad (\mathrm{Ci})$$

$$\frac{G \vdash t \leqslant u}{G \vdash \mathrm{coerce}(t, u)} \quad (\mathrm{Ce})$$

$$\frac{G \vdash t \leqslant u, \qquad t = \langle c\ \ldots\rangle, \qquad u = \langle d\ \ldots\rangle, \qquad c \not\leqslant u}{G \vdash \bot} \quad (\mathrm{C}\bot)$$

$$\frac{G \vdash \langle c\ t_1 \ldots t_n\rangle \leqslant X \qquad X \restriction S \leqslant u \qquad c \in S}{G \vdash \langle c\ t_1 \ldots t_n\rangle \leqslant u} \quad (\mathrm{F})$$

then for any $t' = \langle c'\ \ldots\rangle$, where $c' \in S$, we also have $G \vdash t' \leqslant X$ and thus $G \vdash t' \leqslant Y$. Now, we definitely do not want to create new type expressions out of thin air (as it would endanger the termination of the constraint checking algorithm). Also, intuitively it makes sense that if one type can pass a filter, a more general type should also pass the filter.

4.5 Putting everything together

Let the set of *type expressions* $t, u \in \mathsf{TExp}$ be the minimal set such that:

(1) $\mathsf{TVar} \subseteq \mathsf{TExp}$, and
(2) $\langle c\ t_1 \ldots t_n\rangle \in \mathsf{TExp}$, where n is the arity of c, and $t_i \in \mathsf{TExp}$, for $i \leq n$.

Let the set of *constraints* $\varphi \in \mathsf{Constraint}$ be formulas of the following forms:

(1) $t \leqslant u$, for $t, u \in \mathsf{TExp}$,
(2) $X \restriction S \leqslant t$, for $X \in \mathsf{TVar}$, $t \in \mathsf{TExp}$, and S an up-closed set of constructors, and
(3) $\bot$.

We give the derivation rules for extended constraints in Figure 9 and the type rules for the extended lambda calculus in Figure 10. A filter expression $X \restriction c$ is a shorthand for $X \restriction S$, where S is the smallest up-closed set containing c. Similarly, we use $X \setminus c$ as a shorthand for $X \restriction S$, where S is the largest up-closed set *not* containing c.

Fig. 10. Subtyping rules for extended lambda calculus.

$$\frac{(x : t) \in \Gamma}{\Gamma \Vdash x : t} \qquad \text{(axiom)}$$

$$\frac{\Gamma\,[x \mapsto t] \Vdash M : u}{\Gamma \Vdash \lambda x.M : \langle c_\lambda\ t\ u\rangle} \qquad \text{(abstraction)}$$

$$\frac{\Gamma \Vdash M : \langle c_\lambda\ t\ u\rangle \qquad \Gamma \Vdash N : t}{\Gamma \Vdash MN : u} \qquad \text{(application)}$$

$$\frac{\Gamma \Vdash M_i : t_i, \quad 1 \le i \le n}{\Gamma \Vdash \langle c\ M_1 \dots M_n\rangle : \langle c\ t_1 \dots t_n\rangle} \qquad \text{(constructor)}$$

$$\frac{\begin{array}{c}\Gamma \Vdash M : t\\ t \leqslant X\\ X \restriction c \leqslant \langle c\ u_1 \dots u_n\rangle\\ \Gamma\,[x_1 \mapsto u_1, \dots, x_n \mapsto u_n] \Vdash N : w\\ X \setminus c \leqslant Z\\ \Gamma[y \mapsto Z\,] \Vdash P : w\end{array}}{\Gamma \Vdash \mathsf{case}(M, \langle c\ x_1 \dots x_n\rangle \Rightarrow N, y \Rightarrow P) : w} \qquad \text{(case)}$$

$$\frac{\Gamma \Vdash M : t \qquad t \leqslant u}{\Gamma \Vdash M : u} \qquad \text{(subsumption)}$$

4.6 The extended type checking algorithm

We are now ready to define the extended type checking algorithm. The derivation rules are given in Figure 9. Note that conversion of type expression is defined using the function coerce, as discussed in Section 4.4.

From the rules it is easy to see that a constraint should be *immediately inconsistent* if it is $\perp$ or of the form $\langle c\ \dots\rangle \leqslant \langle d\ \dots\rangle$, where $c \not\leqslant d$.

The computation of G^* needs to maintain two derivation rules; Rule (Ce) and Rule (F) which propagates filters.

Definition 4.1. Given a constraint system G, define $(G)_n$, for $n \ge 0$, to be the smallest sets that satisfy the following:

(1) $(G)_0 = G$.
(2) For all n, $(G)_n \subseteq (G)_{n+1}$.
(3) If $(G)_{n-1}$ contains the constraint $t \leqslant u$, where t and u are constructor expressions, and coerce(t, u) is defined, then coerce$(t, u) \subseteq (G)_n$.
(4) If $(G)_{n-1}$ contains $t \leqslant X$ and $X \leqslant u$ then $(G)_n$ contains $t \leqslant u$.
(5) If $(G)_{n-1}$ contains the constraints $\langle c\ t_1 \dots t_m\rangle \leqslant X$ and $X \restriction S \leqslant u$, where $c \in S$ then $(G)_n$ contains the constraint $\langle c\ t_0 \dots t_m\rangle \leqslant u$.

Let $G^* = \bigcup_n (G)_n$.

The proof has the same structure as the proof sketched in Section 2.

5 HOW TO MAKE ERLANG STATICALLY TYPED

The type rules of the subtyping system are more general than those of Hindley-Milner typing and thus the subtyping system should be able to type any program typable in Hindley-Milner, by simply removing data type definitions and using predefined constructors instead of those given in data type definitions.

The subtyping system should in principle be able to type check a complex program, relying only on top-level specifications and deducing internal data types. In practice, it is probably a good idea to introduce function specifications and data types declarations for various intermediate function definitions and data types as this will help locating the sources of type errors and speed up type checking.

5.1 Type definitions and function specifications

The system accepts source files containing Erlang code, type definitions and function specifications. One example:

```
-module(example1).
%: +type list(X) = [] + [X|list(X)].

%: +func append :: list(X) * list(X) -> list(X).
append([A | B], C) ->
    [A | append(B, C)];
append([], C) -> C.

%: +func dup :: list(integer()) -> list(integer()).
dup(S) ->
    append(S, S).
```

(In Erlang, lower case identifiers without arguments indicate atoms, upper case variables are variables.) The first type declaration defines the polymorphic and recursive type `list()`, which is of course either the empty list constructor (`[]`) or the cons constructor applied to the type parameter and the list type, (`[X|list(X)]`).

The character combination “`%:`” is treated as white space by the type system’s scanner while the Erlang compiler treats any sequence of characters that begins with `%` as a comment. Thus definitions and specifications will be read by the type checker but ignored by the compiler.

Type definitions use the keyword `type`. The type specification of the function `append` simply states that the function takes two lists as arguments, and returns a list of the same type.

In the language for types used in specifications and type definitions, an atom followed by an argument list (for example, `list(integer())` as in the example above) indicates either a type constructor or a type defined in some type definition. Some constructors use different syntax, for example the empty list `[]` and the list constructor `[...|...]`. Also, atoms and tagged and untagged tuples have the same syntax as in Erlang. As Erlang allows a function to have any number of arguments, we use a function type constructor for each number of arguments. Examples of function types with zero, one, or two arguments:

```
() -> atom()   integer() -> atom()  integer() * float() -> atom().
```

Among other primitive types in the source language are `atom()`, the type of atoms, and `integer()`, the type of integers.

Finally, we use the notation `any()` and `none()` for the universal and empty types, respectively.

5.2 Unsafe features

5.2.1 Promises. Many functions in the standard library are ill-suited for static typing. For example, there are many functions that may return a value of any type. Among these are primitives for process communication and functions that read data from a file or from standard input.

Rather than barring programmers from using such operations, our system includes a primitive `promise` that allows the programmer to assert that a variable has a particular type. We illustrate the use of the primitive with a simple example.

```
%: +func f::() -> integer().
f() ->
    {ok, X} = io:read(">"),
%:  promise X :: integer(),
    X.
```

Now, promises are unsafe in the sense that if the programmer lies to the type system in a promise the type system will trust the promise. A cautious programmer could of course insert code that checked the promise, and in a more well-integrated system such tests could be inserted automatically.

The implementation of the type system uses promises in four locations. Two uses occur in the module `program` and are associated with calls to the function `get_value` of the library module `proplists`, which extracts a field from a property list. Since a property list may store any value, and different types of values are associated with different properties, there is no way to statically determine the type of one particular field.

The two other uses of `promise` occur in the module `record_expand` which expands records. This use of a promise could perhaps be avoided by more careful coding and better use of polymorphism.

5.2.2 External modules. The type system checks one module at the time. If a second module is referenced, and specifications are available, the type system will under default settings use the specifications instead of analysing the second module. Naturally, until the second module is also checked, there is no way of knowing whether the specification in the second module really conforms with the actual code.

In the type system, there are some places where typing relies on specification files, but the type system has not checked that the specifications match the corresponding function definitions. The parser which is based on the standard Erlang parser is not checked. Instead, the abstract syntax tree which is generated by the parser is specified separately. Also, there is a module that implements a modified version of the standard Erlang preprocessor. The type system relies on specifications of three functions of that module that are not checked. There are also seven functions in standard libraries (involving IO, the file system and timers) that are not checked. Perhaps more importantly, the module `lists` which implements various operations on lists could not be checked. The reason is that many functions in that library manipulate lists of tuples, for example `keysort`, which takes an integer and a list of tuples. The list is sorted by the position given by the integer. To type programs that use the function, a precise specification of this function should reflect not only that the second argument and the result are both lists of tuples but also that the tuples are of the *same* type as the input tuple.

5.2.3 Calls with unknown destinations. One feature that makes Erlang programs hard to analyse is the ability to write calls whose destination is determined at run time. One can even write code that could call any exported function in any module, depending on data read from a file or standard input.

The type checker allows calls where the destination cannot be determined but do not attempt to track their destination. Since the destination may be any exported function in any module it is impossible to guarantee that a function called will not have a specification that the call breaks. Thus, whenever a call occurs where the destination cannot be determined the responsibility for ensuring that the call does not break any specification falls on the programmer. (The implementation of the type system does not contain any calls where the destination cannot be determined.)

6 THE IMPLEMENTATION

6.1 Predicates

The front end of the type checker translates function definitions, type definitions and specifications into *predicates.*

After the predicates have been generated, the remaining phases of the type checker do not rely on any other information beside the structure of the predicates and the constraint systems explicit in the predicates.

A predicate takes the form

$$\text{predicate } \mathit{Name}\ [X_1, ..., X_n]\ \mathit{Body}$$

where *Name* is an identifier, $X_1, \ldots, X_n$ are variables (the parameters of the predicate), *Body* is a set of constraints and calls. A *call* is of the form

$$\text{call } \mathit{Name}\ [Y_1, \ldots, Y_m]$$

We refer to $Y_1, \ldots, Y_m$ as the *arguments* of the call. Free variables of *Body* that are not among the parameters are implicitly for-all quantified. The resolution of a call is simple; the call is replaced with the body of the predicate where every variable that occurs as a parameter is replaced with the corresponding argument, and other free variables are replaced with fresh variables.

6.1.1 Examples. We show predicates for simple type definitions. First, a simple type definition with two alternatives:

```
    +type bool() = true + false.
```

Type definitions are used in two situations, when generating a type, and when checking that a supplied type is indeed as specified. Since these two cases require different constraints and don't interact we define two predicates for each type definition, a *lower* predicate and an *upper* predicate.

Consider first the lower predicate for the type `bool()`.

$$\begin{aligned} &\text{predicate } \mathit{type\_lower\_bool}\ [T] \\ &\quad ([] \to A) \leqslant T, && (3) \\ &\quad \langle \textbf{atom}_{\texttt{true}} \rangle \leqslant A, && (4) \\ &\quad \langle \textbf{atom}_{\texttt{false}} \rangle \leqslant A. && (5) \end{aligned}$$

Like all predicates for type definitions, the predicate takes a single parameter (T). As the type `bool` does not take any parameters, the predicate generates a function type without parameters (3). The result type (A) describes the possible values of a variable or expression of type `bool()`. There are two possible values, the atom `true` or the atom `false`.

Next, the upper predicate for `bool()`.

$$\begin{aligned} &\text{predicate } \mathit{type\_upper\_bool}\ [T] \\ &\quad ([] \to A) \leqslant T, && (6) \\ &\quad A \restriction \textbf{atom}_{\texttt{true}} \leqslant \langle \textbf{atom}_{\texttt{true}} \rangle, && (7) \\ &\quad A \setminus \textbf{atom}_{\texttt{true}} \leqslant \langle \textbf{atom}_{\texttt{false}} \rangle. && (8) \end{aligned}$$

In the upper predicates for `bool()`, we use filters (as explained in Section 4.2) to isolate the two cases. When filtered with the set $\{\textbf{atom}_{\texttt{true}}\}$ the type of A must be a subtype of $\langle \textbf{atom}_{\texttt{true}} \rangle$ (7). The third line (8) uses a filter to exclude any use of the atom `true`. If A is not the atom `true`, the type of A must be a subtype of the atom `false`.

Next we consider a simple parametric type.

```
    +type option(X) = none + {some, X}.
```

Both the lower and upper predicate define the type of T as a function with one parameter. In the lower predicate, the parameter X is introduced in constraint (9) and used in constraint (11). By constraint (10) the result may be the atom none and by constraint (11) the result may be a tuple, where the second element is given by the parameter X.

$$\begin{aligned} &\text{predicate } \mathit{type\_lower\_option}\ [T] \\ &\quad ([X] \to A) \leqslant T, && (9) \\ &\quad \langle \textbf{atom}_{\mathsf{none}} \rangle \leqslant A, && (10) \\ &\quad \langle \textbf{tuple}^2_{\mathsf{some}}\ X \rangle \leqslant A. && (11) \end{aligned}$$

The upper predicate follows a similar pattern. The type passed to the predicate needs to be a function type with one parameter, as the type is parametric (12). Filters are used to distinguish between the cases when the result of the supplied type is the atom none (13) and when it is *not* (14).

$$\begin{aligned} &\text{predicate } \mathit{type\_upper\_option}\ [T] \\ &\quad T \leqslant ([X] \to A), && (12) \\ &\quad A \restriction \textbf{atom}_{\mathsf{none}} \leqslant \textbf{atom}_{\mathsf{none}}, && (13) \\ &\quad A \setminus \textbf{atom}_{\mathsf{none}} \leqslant \langle \textbf{tuple}^2_{\mathsf{some}}\ X \rangle. && (14) \end{aligned}$$

We end with a (non-parametric) recursive type,

```
    +type intlist() = [] + [integer() | intlist()].
```

Both the lower and upper predicate are recursive. In the lower predicate, the recursive call supplies the type of the rest of the list (19).

$$\begin{aligned}
&\text{predicate } \textit{type\_lower\_intlist}\ [T] \\
&\qquad ([] \to A) \leqslant T, && (15)\\
&\qquad \langle \textbf{nil} \rangle \leqslant A, && (16)\\
&\qquad \langle \textbf{cons}\ \langle \textbf{integer} \rangle\ B \rangle \leqslant A, && (17)\\
&\qquad \text{call } \textit{type\_lower\_intlist}\ [U], && (18)\\
&\qquad U \leqslant ([] \to B). && (19)
\end{aligned}$$

In the upper predicate, constraint (24) gives an upper bound to the rest of the list. The next section will describe how recursive predicates are replaced with constraints.

$$\begin{aligned}
&\text{predicate } \textit{type\_upper\_intlist}\ [T] \\
&\qquad T \leqslant ([] \to A), && (20)\\
&\qquad A \restriction \textbf{nil} \leqslant \langle \textbf{nil} \rangle, && (21)\\
&\qquad A \setminus \textbf{nil} \leqslant \langle \textbf{cons}\ \langle \textbf{integer} \rangle\ B \rangle, && (22)\\
&\qquad \text{call } \textit{type\_upper\_intlist}\ [U], && (23)\\
&\qquad ([] \to B) \leqslant U. && (24)
\end{aligned}$$

6.2 Recursion in predicates

Let's first look at the case where a predicate is recursive, but there are no mutually recursive predicates. Consider the predicate *type_lower_intlist* of the previous section. It contains a call call *type_lower_intlist* $[U]$. The strategy is simply to merge the parameters with the arguments in the recursive calls, i.e., replace all of them with a single variable. The recursive calls can now be removed. In the example, this gives us the following predicate:

$$\begin{aligned}
&\text{predicate } \textit{type\_lower\_intlist}\ [T] \\
&\qquad ([] \to A) \leqslant T, && (25)\\
&\qquad \langle \textbf{nil} \rangle \leqslant A, && (26)\\
&\qquad \langle \textbf{cons}\ \langle \textbf{integer} \rangle\ B \rangle \leqslant A, && (27)\\
&\qquad T \leqslant ([] \to B). && (28)
\end{aligned}$$

Now, it should be stressed that this approach to recursion will sometimes be overly aggressive; it is possible to create a recursive predicate where merging recursive calls in this manner gives a non-recursive predicate where the body is inconsistent, but where a more conservative approach would have avoided inconsistency. However, as our approach generalises Hindley-Milner typing, it seems safe to assume that it will work well in practice.

To handle mutual recursion, it is useful to view the predicates as a directed graph where each predicate is a node and an edge connects two predicates if there is a call in the first predicate to the second. Mutually recursive predicates form strongly connected components.

Predicates that form a strongly connected component are combined into a new predicate. The parameter list of this predicate is the concatenation of the parameter lists of the predicates it replaces. Any call to one of the predicates in the strongly connected component is replaced with a call to the new predicate, where the argument list is adjusted to take into account the position in the new predicate.

Non-recursive calls between predicates are treated as polymorphic. Thus each such call results in the duplication of constraints. To reduce the cost of duplication, various simplification algorithms are applied before duplication.

6.3 The constraint solver

The constraint solver uses a graph representation where nodes are type variables and edges are labeled with filters and represent constraints of the form $X \restriction S \leqslant Y$. With each node, say for a variable X, we associate constructor expressions t such that $t \leqslant X$ (*supports*) and $X \leqslant t$ (*covers*). As suggested by Heintze and Tardieu (2001), we do not compute a representation of the transitive closure. Instead, when a link $X \restriction S \leqslant Y$ is added, a depth first search collects the direct and indirect supports of X and a second dfs collects the covers of Y. The covers and supports are then combined.

The solver (and the rest of the type checker) is written in a pure functional style, with the exception of IO and calls to the `timer` library.

6.4 Constraint simplification

Since our implementation of polymorphic type checking sometimes requires a constraint system to be duplicated, it reasonable to use constraint simplification to (hopefully) improve performance. We have developed two approaches to constraint simplification.

Our starting point is a constraint system G and the set of variables P which serve as an interface to the constraints in G. The constraint system G represents a definition of a function or a type definition. The constraint system needs to be duplicated if it is used in different contexts.

In the first simplification, we consider *reachability*, i.e., the set of constraints in G that can be reached from P.

The second simplification considers *stability*. Given a constraint system G and a set of visible variables P it sometimes happens that a variable is reachable, but that there is no need to duplicate the variable if the constraint system is duplicated. Suppose, for example:

$$P = \{X\}, G = \{\langle \mathbf{cons}\ Y\ Z\rangle \leqslant X, \langle \mathbf{cons}\ Y_1\ Y_2\rangle \leqslant Y\}. \tag{29}$$

Even if G is used in different contexts, there is no need to duplicate the variables Y, Y_1 and Y_2. This situation occurs for example if G is the constraint system of a function that returns a complex data structure but the data structure does not depend on the input. Obviously, the type of the result will always be the same.

7 MEASUREMENTS

Table 2 summarises the source files examined in the measurements and the results.

In addition to the source modules listed in the table, there are a few files containing only type declarations and specifications. The largest is absyn.espec (112 lines), containing type declarations describing the abstract syntax tree generated by the parser.

The module poly serves as top level, coordinates some tasks and also implements polymorphism as described in Sections 3 and 6.2. Module program stores the source code in an intermediate form, supplied by convert and where record_expand has expanded any use of records. Module pos helps in the introduction of temporaries by tracking the position in a program, this is used both in convert and in the later translation to constraints. sanity traverses the code and checks that all referenced definitions and specifications have been loaded by program. agenda takes a program and a list of specified functions and creates a list of definitions and specifications (an agenda) that need to be translated into constraints. The module walker takes an agenda and generates predicates (see Section 6.1).

The module match generates constraints from pattern matching but as it contains no specifications the module has been measured with walker, which is the only other module that calls it. The module poly will then merge mutually recursive predicates,relying on graph for detecting strongly connected components, and the module coalesce for coalescing them. poly will then expand predicate calls, perhaps interleaved with constraint simplification (module reach).

The module solver implements the constraint solver (Section 6.3) relying on module conn for various properties of constructors, for example variance and the function coerce (Section 4.4) which implements conversion. It should be noted that both solver and reach are independent of the set of constructors used by the type checker. The modules rfilter implements filters, i.e., sets of constructors. Finally, worklist supports solver in the scheduling of tasks.

The type checker was set to use constraint simplification and "prefer specifications", i.e., to use specifications of functions, when available, to check the outcomes of recursive calls.

All measurements have been run on a 1.3 GHz Intel Core i5 (a 2013 Macbook Air). In the measurements, only one core was used. The Erlang implementation used a BEAM byte-code emulator.

8 RELATED WORK

Kozen et al. (1994) showed that the problem of checking that an term in lambda calculus can be typed by a subtyping system could be solved in $O(n^3)$ time.

Palsberg and O'Keefe (1995) show that it is possible to use *control flow analysis* to check that a lambda term can be typed under a subtyping system. Their language is richer than that of Kozen et al. (it has a universal type, an empty type and a primitive type distinct from function types). It is not clear that Kozen's approach could be used to type a language with these features. On the other hand, even though Palsberg's language is quite limited, the

Table 2. Modules in the type checker. LOC: lines of code. LOD: lines of specifications and declarations. Blank lines and comments are not counted. The final column shows time to check the module.

Module	LOC	LOD	Time
agenda	239	15	1.50
coalesce	269	11	0.96
conn	98	10	0.15
convert	943	103	16.11
graph	168	13	0.12
poly	362	23	7.26
pos	18	6	0.01
program	372	49	9.84
reach	501	26	2.00
record_expand	119	22	1.39
rfilter	343	15	0.75
sanity	237	12	0.77
scfa_file	60	5	0.05
solver	508	67	2.09
walker+match	992	67	4.27
worklist	39	8	0.04

approach requires several representations and transformations, and careful proofs that the transformations preserve relationships.

Eifrig et al. (1995) present an interesting approach to subtyping. Types are *constrained*, i.e., each type comes with a constraint system which gives a rich type system, though it does not seem that this gives any additional expressiveness compared to the approach in this paper. Like this paper, Eifrig uses a propagation algorithm to determine whether a constraint system is acceptable. Unlike this paper, there is no attempt to link the propagation algorithm to a definition of consistency using derivation rules, instead they give a subject reduction proof where they show that each reduction step preserves the outcome of the propagation algorithm. This is unsatisfactory from a theoretical point of view, a practical problem is that it makes the type system hard to extend; any modified version of the type system requires a new algorithm for checking constraints. Since the proof of the subject reduction property depends on the algorithm every new version of the algorithm needs a new version of the (rather tedious) proof. Any mistakes in the design of the algorithm will become apparent at a late stage, and it will be hard to tell whether the problem is due to a mistake in the design of the algorithm or in the underlying type system.

Marlow and Wadler (1997) describe an early prototype of a static type system for Erlang written in Haskell and report very promising results; the type system has been applied to thousands of lines of library code and no difficulties are antipicated. However, Erlang has many features that the type system should not be able to handle, and even “nice” functional programs sometimes do things that should be hard to express in their type system. Their constraint language uses a form of discriminated unions that give about the same expressiveness as the filter concept described in Section 4.2. Like the type system described by Eifrig et al. above, consistency is defined using an algorithm instead of by a set of derivation rules. This would make extending the system difficult and error-prone.

Typed Scheme (Tobin-Hochstadt and Felleisen 2008) requires that the program contains type specifications of all functions and data structures. Thus, the problem of type checking is in some regards much simpler as there is only a limited need to deduce types for immediate values. In contrast, the system presented here can deduce complex intermediate data structures.

There are several other recent attempts to integrate static and dynamic typing that rely on some form of subtyping but not in combination with type inference, for example (Flanagan 2006; Knowles and Flanagan 2010; Siek and Garcia 2012; Siek and Taha 2006; Wadler and Findler 2009; Wrigstad et al. 2010). These systems rely on run-time type checks in the conversion from dynamically typed values to values with static types.

Dolan and Mycroft (2017) present a subtype system for SML, extended with a universal type, an empty type and a record concept. Interestingly, they report that their type system has principal types. However, as the principal types are not minimal (in fact, a function may have an infinite set of principal types of unbounded size) the advantage of principal types is unclear. Their implementation of polymorphism relies on heuristic simplification of principal types, analogous to the constraint simplification algorithms exploited in this paper.

Most type systems (including the one presented in this paper) attempt to guarantee some degree of safety from type errors at run-time. Lindahl and Sagonas (2006) take the opposite approach and give a type system for Erlang that that only rejects programs that are *guaranteed* to fail. This allows the type system to work with programs that were not written with static typing in mind.

9 CONCLUSIONS

Designing a static type system for a programming language that was not designed for static typing poses many challenges. Sometimes typing a program requires some minor adjustments, sometimes there are features that seem fundamentally unsuited for static typing. More interesting are situations that seem amenable to static typing, if only the type system was a little bit more powerful.

The subtyping system we have developed is a generalisation of Hindley-Milner type inference. As Hindley-Milner type inference has been used in functional programming for decades, we expected that a generalisation should be capable of handling most functional programs that did not involve any exotic features of Erlang. Experience has confirmed this expectation; programs that would have typed in, say, an SML implementation will indeed type here.

The interesting question is: which programs can be typed under a subtyping system that cannot be typed by the Hindley-Milner system? There are some obvious situations. For example: a complex type that uses fewer constructors than another and is thus a subtype or a type that uses the same constructors as another (but is otherwise unrelated). The use of the subtyping system in the typing of the implementation of the subtyping system has offered some insight in the practical aspects of using subtyping in development.

REFERENCES

Roberto M Amadio and Luca Cardelli. 1993. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 15, 4 (1993), 575–631.

Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language.* Ph.D. Dissertation. University of Copenhagen.

Henk Barendregt, Wil Dekkers, and Richard Statman. 2013. *Lambda calculus with types.* Cambridge University Press.

Stephen Dolan and Alan Mycroft. 2017. Polymorphism, Subtyping, and Type Inference in MLsub. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017).* ACM, New York, NY, USA, 60–72. DOI : http://dx.doi.org/10.1145/3009837.3009882

Jonathan Eifrig, Scott Smith, and Valery Trifonov. 1995. Type inference for recursively constrained types and its application to OOP. *Electronic Notes in Theoretical Computer Science* 1 (1995), 132–153.

Cormac Flanagan. 2006. Hybrid Type Checking. In *POPL'06.* ACM, New York, NY, USA, 245–256.

Cormac Flanagan and Matthias Felleisen. 1999. Componential Set-based Analysis. *ACM Trans. Program. Lang. Syst.* 21, 2 (March 1999), 370–416. DOI : http://dx.doi.org/10.1145/316686.316703

Nevin Heintze. 1994. Set-Based Analysis of ML Programs. In *ACM Conference on Lisp and Functional Programming.* 306–317.

Nevin Heintze and Olivier Tardieu. 2001. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *Programming Language Design and Implementation (PLDI).* 254–263.

Leon Henkin. 1949. The Completeness of the First-Order Functional Calculus. *Journal of Symbolic Logic* 14, 3 (1949), 159–166.

Neil D. Jones. 1981. Flow analysis of lambda expressions. In *Automata, Languages and Programming.* 114–128.

Kenneth Knowles and Cormac Flanagan. 2010. Hybrid Type Checking. *ACM Trans. Program. Lang. Syst.* 32, 2, Article 6 (Feb. 2010), 34 pages.

Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. 1994. Efficient inference of partial types. *J. Comput. System Sci.* 49, 2 (1994), 306–324.

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

Harry G. Mairson. 1990. Deciding ML Typability is Complete for Deterministic Exponential Time. In *POPL'90.* ACM, New York, NY, USA, 382–401. DOI : http://dx.doi.org/10.1145/96709.96748

Simon Marlow and Philip Wadler. 1997. A practical subtyping system for Erlang. *ACM SIGPLAN Notices* 32, 8 (Aug. 1997), 136–149.

Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17 (1978), 348–375.

John C. Mitchell. 1984. Coercion and Type Inference. In *Principles of Programming Languages.* ACM, 175–185.

John C. Mitchell. 1991. Type inference with simple subtypes. *Journal of Functional Programming* 1 (1991), 245–285.

Jens Palsberg and Patrick O'Keefe. 1995. A type system equivalent to flow analysis. *ACM Toplas* 17, 4 (July 1995), 576–599.

Benjamin C. Pierce. 2002. *Types and programming languages.* MIT press.

François Pottier. 2001. Simplifying subtyping constraints: a theory. *Information and Computation* 170, 2 (2001), 153–183.

John C. Reynolds. 1968. Automatic Computation of Data Set Definition. In *Proceedings of the IFIP Congress.* 456–461.

Olin Shivers. 1988. Control Flow Analysis in Scheme. In *PLDI'88.* 164–174.

Jeremy G. Siek and Ronald Garcia. 2012. Interpretations of the gradually-typed lambda calculus. In *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming.* ACM, 68–80.

Jeremy G. Siek and Walid Taha. 2006. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, Vol. 6. 81–92.

Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The design and implementation of typed scheme. *ACM SIGPLAN Notices* 43, 1 (2008), 395–406.

Dirk van Dalen. 2013. *Logic and structure, fifth edition.* Springer-Verlag.

Philip Wadler and Robert Bruce Findler. 2009. Well-typed programs can't be blamed. In *Programming Languages and Systems*. Springer, 1–16.

Andrew K. Wright and Matthias Felleisen. 1994. A syntactic approach to type soundness. *Information and computation* 115, 1 (1994), 38–94.

Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. 2010. Integrating Typed and Untyped Code in a Scripting Language. In *POPL'10*. New York, NY, USA, 377–388. DOI: http://dx.doi.org/10.1145/1706299.1706343